Specification and Analysis of Component-Based Software in a Concurrent Setting

Sotiris K Moschoyiannis

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

Centre for Software Systems Engineering
Department of Computing
School of Electronics and Physical Sciences
University of Surrey
 Guildford, Surrey GU2 7XH, U.K.

September 2005

© Sotiris K Moschoyiannis 2005
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 and distributed applications, pose greater challenges. This thesis provides a formal framework for managing the dependencies between components, in terms of their interactions in a concurrent setting.

In our approach, composites and single components are represented by a component signature, which identifies a component, and a vector language, also called component language, which describes the behaviour of a component. This language-based representation of component behaviour makes it possible to capture concurrency at both the individual component level and the composition level. The interpretation of concurrency is that of a non-interleaving model, with the notion of causal independence lifted to vectors. We describe how component languages are obtained from scenario-based specifications, typically used in an industrial context. Based on the order structure of a component language, we identify implicit or missing interactions which represent potentially faulty or simply unthought scenarios. This excludes pathological behaviour, the source of which can be traced back to inconsistencies in the sequence diagrams of the scenario specification such as race conditions, and this gives a characterisation of well-behaved components.

Components are put together in our approach by matching required and provided interfaces in terms of the respective sequences of events. This builds on the concept of parallel composition in process algebras. We show that the properties that define well-behaved components are preserved under composition in the resulting composite. Well-behaved components give rise to discrete behavioural presentations which can capture concurrency and simultaneity between event occurrences on component interfaces. Well-behaved components are also associated with automata whose transition structure reflects the concurrency in the corresponding component language. This state-based description of component behaviour is graphically represented using state diagrams.

This formal framework for components has been related to more conventional approaches to software design, as exemplified by strong connections to UML. It can aid designers in determining the complete set of intended behaviours before generating state models of the scenario-based specifications.

Key words: components, scenarios, interactions, concurrency, composition, formal languages, vector semantics, order theory, discreteness, left-closure, primes, automata
Acknowledgements

I would like to express my gratitude to Mike Shields for exposing me to models of concurrency. Discussions with Mike have always been stimulating and thought-provoking. Even when the discussion would go all over the place it would always somehow return to concurrency, and on occasion end with Mike clapping his hands or banging his desk to explain the varying interpretations! I wish he continues to be a true academic for many years to come. I am grateful to Professor Paul Krause for his help and guidance with many discussions, including industrial case studies. There is a lot to learn from him at both a professional and a human level. My very special thanks go to Juliana Küster-Filipe for finding the time to discuss my research and suggest improvements. Her encouragement and guidance in formalising interactions are gratefully acknowledged. She exposed me to LSCs and offered a pleasant break from isolation while working on this thesis. I have also benefited from discussions with Bill Mitchell, including case studies, and David Pitt. Thanks are also due to Ralf Reussner for his suggestions and encouragement during workshops and via email. I would also like to thank the Department of Computing for the financial support while working on this thesis.

My warmest thanks go to Maria Spyropoulou for supporting me in so many different ways and being a wonderful distraction for the best part of this thesis. Maryrinal!

Special thanks go to my family for being my family, and my sister Xenia for just being my big sister.

I feel a need to thank A (she knows) but I don't really know why. That is future work.

Finally, I owe so much to my grandfather Sotiros K. Moschoyiannis and this thesis is dedicated to him.
# Contents

1 Introduction
1.1 Setting the Context .......................... 1
1.2 Main Objectives of the Thesis .............. 5
1.3 Plan of the Thesis ............................ 6

2 Approaches to Modelling Component-Based Software
2.1 Basic Component Concepts .................. 10
2.2 Graphical Descriptive Approaches ......... 15
  2.2.1 The Koala Model for components ....... 15
  2.2.2 UML for components ................. 20
  2.2.3 Scenario-based descriptions .......... 37
2.3 Formal Approaches .......................... 48
  2.3.1 Algebraic approaches .................. 51
  2.3.2 Logic-based approaches .............. 59
  2.3.3 Automata-based approaches .......... 63
2.4 Concluding note ............................. 71

3 A Formal Language for Components .......... 75
3.1 Formalisation of a Component ............ 76
  3.1.1 Component Signature .................. 79
  3.1.2 Component Language .................. 82
3.2 From UML to the Component Language .... 88
  3.2.1 Sequence diagrams in UML .......... 88
  3.2.2 Sequence diagrams for components .. 94
  3.2.3 Sequential locations ................. 101
3.2.4 Alternative locations .................................................. 107
3.2.5 Parallel locations ......................................................... 113
3.2.6 Simultaneous locations .................................................. 121
3.2.7 Nested locations ........................................................... 125
3.2.8 Obtaining the component language ................................. 131
3.3 Illustration by example .................................................... 134
3.4 Concluding note .............................................................. 149

4 Properties of a Component Language ........................................ 153
4.1 Order Theoretic Properties .................................................. 154
4.2 Well-behaved Components ................................................ 160
4.3 Relations between Events in a Component Language .............. 167
  4.3.1 Events in sequence ..................................................... 167
  4.3.2 Concurrent events ...................................................... 169
  4.3.3 Events in conflict ...................................................... 176
4.4 Illustration by example .................................................... 178
4.5 Concluding note .............................................................. 192

5 Composition of Components .................................................. 195
5.1 Formalisation of Component Composition ............................. 196
  5.1.1 Composing signatures ................................................ 197
  5.1.2 Composing component vectors ................................. 201
  5.1.3 Algebraic properties of composition ............................ 204
5.2 Visualising Composition with UML ..................................... 208
5.3 Well-behavedness of the Composite Component .................... 211
5.4 Illustration by example .................................................... 216
5.5 Concluding note .............................................................. 224

6 Event-oriented Description of Component Behaviour .................. 227
6.1 Behavioural Presentations .................................................. 229
6.2 Discrete Behavioural Presentations ...................................... 234
6.3 From Component Languages to Behavioural Presentations ....... 239
6.4 Illustration by example .................................................... 248
6.5 Concluding note .............................................................. 253
7 State-oriented Description of Component Behaviour .......................... 255
  7.1 Automata for Well-Behaved Components ........................................ 256
    7.1.1 UML state diagrams for \( \Sigma \)-automata ............................... 266
  7.2 From Automata to Component Languages ......................................... 269
  7.3 Composing \( \Sigma \)-automata .............................................................. 273
  7.4 Illustration by example ................................................................. 279
  7.5 Concluding note ............................................................................. 283

8 Concluding Remarks ........................................................................... 287
  8.1 Summary .......................................................................................... 287
  8.2 Future Directions ............................................................................ 291
  8.3 Afterword ........................................................................................ 302

A Nested locations in UML2.0 sequence diagrams: an example ............ 305

B Index of Terms and Symbols ............................................................... 319
List of Figures

2.1 A software component in Koala ................................................................. 16
2.2 Koala's graphical notation ........................................................................ 17
2.3 Horizontal communication in Koala ......................................................... 18
2.4 The package structure of the UML metamodel ......................................... 27
2.5 A software component in UML1.x notation ............................................ 29
2.6 Component specification architecture using UML1.x notation ................. 30
2.7 UML1.x extensions for component specification ..................................... 31
2.8 A software component using UML2.0 notation ....................................... 33
2.9 A structure diagram for connecting components in UML2.0 ................. 34
2.10 A UML2.0 composite structure diagram for component \( AB \) ......... 35
2.11 A UML sequence diagram with a race condition .................................. 38
2.12 A universal LSC ...................................................................................... 39
2.13 The parallel construct in UML2.0 and a possible interpretation ............ 42
2.14 A high-level MSC .................................................................................. 45
2.15 Parallel composition of components in [Bro00] .................................... 54
2.16 Construction of the C-FSM in [SR02a] .................................................... 65

3.1 Graphical representation of a component ................................................ 79
3.2 Notation for component vectors associated location \( l \) ......................... 104
3.3 Example UML 2.0 sequence diagram ...................................................... 105
3.4 Example of an \texttt{alt} interaction fragment in a UML 2.0 sequence diagram 110
3.5 Notation for component vectors of the end location of \texttt{par} ............... 117
3.6 Example of a \texttt{par} interaction fragment in a UML 2.0 sequence diagram 118
3.7 Example of a \texttt{sim} interaction fragment in a UML 2.0 sequence diagram 123
3.8 Example of an \texttt{alt} fragment within another \texttt{alt} fragment .................. 126
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.9</td>
<td>UML 2.0 case study example</td>
<td>135</td>
</tr>
<tr>
<td>4.1</td>
<td>Event b₁ followed by simultaneous events a₂, d₁</td>
<td>168</td>
</tr>
<tr>
<td>4.2</td>
<td>Concurrent events d₁ and e₁</td>
<td>173</td>
</tr>
<tr>
<td>4.3</td>
<td>Events a₁ and a₂ are in conflict</td>
<td>177</td>
</tr>
<tr>
<td>4.4</td>
<td>Events a₁ and c₁ are concurrent in (ii) but not in (i)</td>
<td>178</td>
</tr>
<tr>
<td>4.5</td>
<td>Component specification architecture for a TV platform</td>
<td>179</td>
</tr>
<tr>
<td>4.6</td>
<td>The MANUAL STORE menu at the user level</td>
<td>180</td>
</tr>
<tr>
<td>4.7</td>
<td>Component specification architecture</td>
<td>180</td>
</tr>
<tr>
<td>4.8</td>
<td>The CMenu component performing the tuning task</td>
<td>182</td>
</tr>
<tr>
<td>4.9</td>
<td>Order structure of Vₘ, as given by the designer</td>
<td>184</td>
</tr>
<tr>
<td>4.10</td>
<td>Order structure of Vₘ, as given by the model</td>
<td>186</td>
</tr>
<tr>
<td>4.11</td>
<td>Discreteness of CMenu component</td>
<td>187</td>
</tr>
<tr>
<td>4.12</td>
<td>Local left-closure of CMenu component</td>
<td>188</td>
</tr>
<tr>
<td>4.13</td>
<td>Forbidden behaviour of the CMenu</td>
<td>189</td>
</tr>
<tr>
<td>5.1</td>
<td>Composition of CMenu and CTuner</td>
<td>209</td>
</tr>
<tr>
<td>5.2</td>
<td>Component specification architecture for the tuning task</td>
<td>217</td>
</tr>
<tr>
<td>5.3</td>
<td>The CTuner component performing the tuning task</td>
<td>219</td>
</tr>
<tr>
<td>6.1</td>
<td>Behavioural presentation model for the CMenu component</td>
<td>252</td>
</tr>
<tr>
<td>7.1</td>
<td>A well-behaved component, with two interfaces</td>
<td>262</td>
</tr>
<tr>
<td>7.2</td>
<td>Concurrent transitions e₁,e₂</td>
<td>262</td>
</tr>
<tr>
<td>7.3</td>
<td>Condition (1) of Definition 7.1.5</td>
<td>263</td>
</tr>
<tr>
<td>7.4</td>
<td>Condition (2) of Definition 7.1.5</td>
<td>264</td>
</tr>
<tr>
<td>7.5</td>
<td>Condition (3) of Definition 7.1.5</td>
<td>264</td>
</tr>
<tr>
<td>7.6</td>
<td>UML notation for concurrent transitions in a Σ-automaton</td>
<td>268</td>
</tr>
<tr>
<td>7.7</td>
<td>UML state diagram for Mₜ</td>
<td>283</td>
</tr>
<tr>
<td>8.1</td>
<td>Event a either happens on its own or simultaneously with c</td>
<td>292</td>
</tr>
<tr>
<td>8.2</td>
<td>Events a and c either happen in sequence or simultaneously</td>
<td>293</td>
</tr>
<tr>
<td>A.1</td>
<td>Example of an alt fragment within a par fragment</td>
<td>306</td>
</tr>
<tr>
<td>A.2</td>
<td>The order structure of the component language Vₚ of component B</td>
<td>317</td>
</tr>
</tbody>
</table>
List of Tables

6.1 Ordering on occurrences of events of the CMenu component . . . . . . 252
Chapter 1

Introduction

Recent years have seen the increasing use of software in many diverse areas of computation 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 services available to other components or the environment via a set of provided interfaces and, possibly, requires services from other components, in order to deliver those promised, via a set of required interfaces. A component supports a provided interface if the component contains an implementation of all operations defined on that interface. If the component requires access to an operation defined on the provided interface of another component, it issues its request through a required interface. Hence, the functionality of a component is made available to other components.
only through its interfaces which hide its implementation details.

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 incorporate new functions and features. Component-based systems 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 components 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 during development of a single product [vO03]). Moreover, component-based software 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 differing 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 consumer electronics and telecommunications industries or other mission-critical software systems,
1.1. Setting the Context

which make stronger 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 at their interfaces, where inconsistencies tend to occur. As a result, software systems built by assembling together prefabricated components sometimes exhibit pathological behaviour. That is, behaviour not intended by the designers but resulting from such inconsistencies.

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 behaviours of the components at their interfaces. In current component technologies, the notion of an interface is associated with the signatures of the operations a component provides to its environment. This semantically thin interface description does not offer enough information to be of value in reasoning about components beyond the interoperability of their interface operations.

A predicate for managing reuse, replacement and composition of components is the ability to precisely describe their behaviour in terms of communications taking place at component interfaces [SM02, Lau01]. A precise description of the observable behaviour of components boils down to capturing the dependencies between provided and required interfaces of interacting components. In this context, a dependency is understood as the reliance of one component on another to support a specific service. Such dependencies between provided and required interfaces 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 Modeling Language (UML) [BRJ99] have been enhanced with component concepts and now include diagrams for representing the structural dependencies between components in a configuration (which component relies on which other in terms of matching provided and required interfaces) and to a limited extent, their behavioural
dependencies in terms of a causal ordering of interactions.

Scenario-based graphical notations [IT96, DH01a, OMG04] are the mainstay of industrial specifications but often contain inconsistencies between the specified causal ordering of events on component interfaces and the order in which events can occur in practice, so-called race conditions. In addition, requirements specifications often contain implicit assumptions based on the embedding of a component in a particular environment, which may no longer be valid when the component is placed in a different context. The root of such inconsistencies is the difficulty of defining concurrent interactions in scenarios.

It transpires that graphical descriptive approaches alone can not adequately describe component contracts precisely. In order to capture the behavioural aspects of dependencies, a component contract needs to be formalised in terms of the sequences of requests the component is capable of servicing through its provided interfaces and the respective sequences of requests the component makes through its required interfaces 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 strand of work in formal methods is concerned with a precise description of the behaviour of communicating systems. Prime examples are Petri Nets [Pet79b], process algebras [Mil80, Hoa85], trace languages [Maz88] and asynchronous transition systems [Shi85, Bed88], and event structures [NPW81]. A common denominator of these theoretical models is the treatment of the phenomenon of concurrency.

Process algebras such as CSP and CCS identify concurrency with nondeterministic interleaving of events. In event structures 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 mild generalisation of the event structures model, the so called behavioural presentations [Shi88], introduces in addition 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 and one occurs immediately after the other.

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 determines the choice between them. In the context of component-based software engineering, 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 Objectives of the Thesis

The object of this thesis is to provide a formal framework for rigorous analysis and reasoning about components, in terms of their interactions and 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 UML.

The formal framework incorporates a range of concurrency theories, though we have opted for a non-interleaving semantic model which draws upon early ideas on vector languages [Shi79]. This representation of component behaviour allows for expressing concurrency at the level of individual components in addition to the usual concurrency arising through composition. It can be used to enhance scenario-based specifications by ensuring that these describe intended behaviour only. This excludes potential instances of pathological behaviour, the source of which can be traced back to the UML sequence diagrams in the initial specification.

Further, the language-based description of a component can be translated into an event-oriented description of behaviour, based on a behavioural presentation, and a state-oriented description of behaviour, based around asynchronous transition systems. The notion of composition of components within the formal framework draws upon parallel composition in process algebras and allows for reasoning about properties of the composite based on properties of the individual components.
Chapter 1. Introduction

The contribution of this work lies in enriching component-based design practices with mathematical methods which can be used to faithfully determine whether components can be fitted together in a useful way, taking into account concurrency at both the individual component level and the composition level. In doing so, it also refines UML design models used in industrial specifications as it identifies missing interactions which represent potentially faulty or simply unthought scenarios. In addition, it should also contribute towards cementing a relationship between formal foundational research and software systems 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 component-based systems both formally and diagrammatically. After an overview of component concepts for specification and analysis, we describe graphical descriptive approaches for component-based design and discuss formal approaches tailored to the specification and analysis of components.

Chapter 3 introduces a formal description of a component in terms of a signature, which describes the interfaces of a component, and a vector language formed over the signature, which describes the behaviour of a component. The language part of a component comprises vectors, essentially tuples of sequences, where each coordinate corresponds to an interface and contains a finite sequence of events that may occur on that interface. These component vectors provide 'snapshots' of component behaviour over time. In component-based software design, we are interested in the intended behaviour rather than all possible behaviours. We advocate a specification technique that uses UML2.0 sequence diagrams to restrict to an appropriate subset of all possible vectors formed over a given signature. This involves mapping each significant point in a sequence diagram onto component vectors. These vectors capture the observable behaviours of the component during execution of the scenario described in the diagram.
1.3. Plan of the Thesis

The resulting set of component vectors comprises the so-called component language. The ideas are applied to an example case study from the telecommunications industry.

In Chapter 4, we examine the order structure of component languages as a means of determining the actual ordering between events occurring on different interfaces of the component. The order theoretic properties give rise to conditions that determine whether the set of behaviours described by a component language is complete or possible, and potentially undesirable, behaviours have been omitted. Missing behaviours may cause pathological behaviour to emerge since they are due to inconsistencies in the sequence diagrams of the scenario-based specification, e.g. race conditions. This leads to the characterisation of well-behaved components, in the sense that such emergent behaviour has been eradicated. This is demonstrated in an example case study from the consumer electronics domain which is used throughout the thesis to illustrate the ideas and constructions.

Having captured the intended behaviour of components and refined their specifications accordingly, we are in a position to consider putting components together. This is done in Chapter 5 which is concerned with the composition of components. Components are composed by connecting matching interfaces of each. This implies certain conditions on the corresponding signatures and component languages. UML structure diagrams are described for visualising composition within our framework. Attention is then given to preservation of well-behavedness under composition. It turns out that this is the case under a compatibility condition which is a fine-grained version of the condition for composing languages. The conditions are explained and component composition is illustrated with the example of the thesis.

Chapter 6 focuses on the patterns of events the component can experience during its correct participation in the scenarios it is involved in. Building on the order theoretic properties of component languages, a well-behaved component is associated with a discrete behavioural presentation. This provides an event-oriented description of component behaviour, which is particularly useful when the component is to be placed in a different context. The construction is illustrated in detail with the example of the thesis.
Chapter 7 provides a state-oriented description of component behaviour. Well-behaved components are associated with automata, the so-called $\Sigma$-automata, which are specifically tailored to reflect properties of the corresponding component language. This state-based description defines a usage protocol state machine for a component. UML compound transitions are given a concrete semantics in terms of our automata-based formalism and UML state diagrams are used for a graphical representation of $\Sigma$-automata. We also discuss composition in relation to automata. The formal construction that associates components with automata is illustrated with the example of the thesis.

Chapter 8 contains some concluding remarks and summarises the work of the thesis and its contributions. Future developments and directions of this research work are also discussed.
Chapter 2
Approaches to Modelling Component-Based Software

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. In [BRJ99] a model is characterised as being a 'good' model if it includes those elements that have a broad effect and omits those minor elements that are not relevant, at least not at the given level of abstraction.

In the context of component-based software 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 review state-of-the-art techniques for modelling component-based software both formally and diagrammatically. 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. Next, we outline graphical descriptive approaches to software design and examine their suitability for the component-based paradigm. 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 component-based software.

2.1 Basic Component Concepts

Unfortunately most software engineering concepts encompass various aspects and this is often a setback for their intuitive definition. The notion of a software component is by its very nature difficult to define precisely. In fact, there is a noticeable variation in the definition of a component within the component-based software engineering (CBSE) community. The root of this variation seems to be that components can be seen from different angles: a component as a piece 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 software component by enumerating its characteristic properties. According to C. Szyperski [Szy97, SGM02] the properties that characterise a software 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 provides / 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 component. 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 the period of their activation. 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 components 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 software component. However, with regard to 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 software 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 views of components as implementations and components as architectural 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 software component infrastructure, software components and a component model. In particular, a component model is intended to enforce global behaviour on how components in a component-based system 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 state in [HC01] 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 Modeling Language (UML) specification [OMG03b] defines a software 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 the ones 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,
2.1. Basic Component Concepts

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 [OMG04], 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.

[Si 98] discusses 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 software component. Being primarily interested in specification and analysis of component-based software 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's COM/.NET [Cor], Sun's Enterprise JavaBeans (EJB) [Mic03] and the CORBA Component Model [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 software 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 interfaces.

It is not feasible to manage change, replacement and the composition of components 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 the 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-conditions, post-conditions 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 in [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 services 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 services provided by other components.

It transpires that the main challenge in component-based software 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 provided and required interfaces in order to be able to describe the respective com-
ponent contracts precisely. Various approaches and methodologies have been devised with varying levels of precision, ranging from pure diagrammatic techniques to formal approaches.

2.2 Graphical Descriptive Approaches

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 graphical 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 and helps focus the discussion on the key aspects of the software system under development.

In computing, models are usually abstract and are increasingly represented using a diagrammatic notation. The prevalent diagrammatic descriptive technique nowadays is the Unified Modeling Language (UML) [Gro05] which is becoming an industrially well-known standard. Other diagrammatic notations such as scenario description languages, which model system functionality in terms of sequences of interactions between parts of the system, are also widely used in industrial specifications. A variant of such languages appears as part of UML. In this section we describe graphical modelling techniques for software design that highlight current trends in industrial 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 that software is expected to combine previously unrelated functions [KFPRS04]. CE products are no longer isolated entities but have become members of complex product-family structures. These structures exhibit a large degree of diversity in product features.
thermore, 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 the 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 software components. The Koala component model and language [vOvdLKM00, vO02b] was developed in response to this challenge and is currently being used in Philips Electronics [Ele] for developing software embedded in a family of television sets. Its primary objective is to facilitate building a large population of products [vO02a] by reinforcing connection technology between components.

In the Koala model, a component interface is a small set of semantically related functions as in COM and Java. Components access all external functionality through required interfaces and offer functionality through provided interfaces. Koala's graphical notation strongly resembles hardware design, in which reuse has taken place for some years. Components are rendered as rectangles, reminiscent of IC chips, 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. The software component depicted in Figure 2.1, using Koala's graphical notation, provides interfaces \( p_1, p_2, p_3 \) and requires interfaces \( r_1, r_2 \).

![Figure 2.1: A software component in Koala](image)

As can be seen in Figure 2.2, when components are interconnected to form a product,
the resulting configuration looks like an electronic circuit. People in the TV domain will readily understand such pictures.

Figure 2.2: Koala's graphical notation

Koala's graphical notation is slightly extended in [vO03] 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 those further away from it (see [vO03] 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] 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 2.3, the horizontal connections between interfaces of
components $B$ and $C$ allow for direct communication between devices along the same signal path while vertical connections are used for connecting components $A$ and $D$, which are controlled by $B$ and $C$, respectively.

![Diagram of horizontal communication in Koala](image)

**Figure 2.3: Horizontal communication in Koala.**

The Koala interface definition, which essentially lists the function prototypes in C syntax, 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 contains all functions of the previous interface plus some additional ones.

To maximise the potential for reuse, configuration-specific information is moved out of the component, 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* interfaces in this case.

Although Koala is an event-based model it does not define some explicit event-handling mechanism (such as event-subscription or multicasting between components). Events are *signalled* through outgoing (required) interfaces, just like in Visual Basic. An event-handling interface is then provided by another component and is connected to the event signalling interface. (In a multi-threaded system, functions in interfaces are called on the thread of the component raising the event, so the the rule of thumb in Koala is that the handling must be quick and non-blocking.)
We pause to make the observation that the notion of interface in Koala is semantically thin in that it comprises no more than the signatures of the operations the component offers to and requires from its environment. This interface notion, although popular, offers little information to be of value in a more rigorous approach to the composition of systems from components. Additional information about component interfaces, possibly in terms of the orderings of the associated signalled events, could be useful during system verification. Having said that, Koala offers a component technology that stimulates the development of largely independent components, including their evolution and code generation, while defining some architectural issues globally. We briefly describe these below.

The Koala language was inspired by COM/ActiveX [Cor] but particular focus is placed on static binding, since most of the connections between components are known at configuration time, and dynamic binding, since high-end products need to allow for the upgrading of components in the future. Also, an explicit notion of required interfaces, missing in COM/ActiveX, is needed for a pure component-oriented approach.

The architectural description language (ADL) aspect of Koala draws upon Darwin [MDEK95] which provides bindings and considers a notion of provided and required interfaces. In addition to these concepts, Koala also supports a way of adding glue code between components (without adding new components) and a mechanism for handling diversity.

Static binding is performed at configuration time via naming conventions and renaming macros which bind functions of a required interface to those of the corresponding provided interface. Koala uses a switch connector, in combination with diversity interfaces, for dynamic (run-time) binding. [vO03] provides a richer set of connectors to handle structural diversity within a configuration in terms of a fork, a switch, a matrix and a source selection connector.

To facilitate reuse, components are parameterised over all configuration-specific information. A parameter in a component interface can be a boolean method or function indicating the mode of operation, e.g. whether a tuner component should operate in frequency mode or in channel mode. The method is implemented in a module which is
part of the configuration. A module can be seen in the configuration of Figure 2.2.

Modules allow the addition of glue code between components and they can be understood as interfaceless components which can be used for slightly different purposes. They can be declared within a component and be connected to its interfaces, or to those of its subcomponents in case of a compound component. They can also be used to implement basic components. Finally, modules may simply be part of the configuration in which case they have a role to play in handling diversity, as discussed before.

2.2.2 UML for components

We start by outlining the basic concepts and constructs behind the Unified Modeling Language (UML) and then discuss it in view of its applicability for modelling component-based software.

According to its initiators [RJB99], UML is a general-purpose visual modelling language that is used to specify, visualise, construct and document the artifacts of software-intensive systems.

It is Unified because it combines commonly accepted concepts from the leading object-oriented methods. It came about as the offspring G. Booch's Booch method, I. Jacobson's Object-Oriented Software Engineering (OOSE) method and J. Rumbaugh's Object Modeling Technique (OMT). Their expectation that a collaboration had the potential of providing new insights and improvements to all three earlier methods seems to have been reasonably met by the resulting modelling language UML and its subsequent evolutions.

It is a Modeling technique because it attempts to simplify reality by representing complex software systems in terms of a set of diagrams, each capturing a different aspect of the system. UML takes an object-oriented view of the software development process and defers from viewing software systems modelling from an algorithmic perspective which was the traditional approach to modelling software systems. Hence, the main building block for the UML models is the object or the class.

It is a Language because it provides a vocabulary (instances of a construct) and the rules
2.2. Graphical Descriptive Approaches

for combining words (connected instances of constructs) for the purpose of meaningful communication. The vocabulary specifies the constituent parts of the UML diagrams while its rules, referred to as well-formedness rules in [OMG03b], specify constraints over attributes and associations defined in the UML's metamodel.

The primary goals for UML, as outlined in [OMG03b], are as follows:

1. to integrate best practices; since its origins UML was intended to encompass a collection of best engineering practices which had already been proven successful in modelling large-scale, complex systems

2. to encourage the growth of the object-oriented market; the industry can benefit by having a standard modelling language which is well-known to a wide circle of developers

3. to support specifications that are language and development process independent; UML can support different programming languages and various development methods without excessive difficulty

4. to provide users with a ready-to-use visual modelling language to communicate; UML comprises a set of core modelling concepts that are generally accepted by current methods and modelling tools. Based on this core set of concepts, users can develop and exchange models to facilitate discussion and reach a common understanding during design. A concern might be raised at this point with regard to whether the UML models have the unambiguous interpretation necessary to fulfill this goal.

5. to provide extensibility and specialisation mechanisms to extend the core concepts; in order to make UML flexible enough to be tailored as new needs emerge for specific domains, extension mechanisms are provided in its specification. The idea is that deviations from the common case for particular application domains are made possible without changing the core concepts of the language.

6. to provide a formal basis for understanding the modelling language; UML provides a formal definition of the static format of the models using a metamodel
expressed in UML class diagrams. In addition, UML expresses well-formedness constraints in natural language together with Object Constraint Language (OCL) expressions (see [WK99] for an introduction to OCL and [OMG03a] for its latest specification). A major concern can be raised here with regard to whether the trade-off between practical usage and precision of the resulting description has been fairly balanced. It seems that UML lacks a precise behavioural semantics in favour of being user friendly and easily applicable in practice. This is further discussed in the concluding note of this chapter.

7. to support high-level development concepts such as components, collaborations, frameworks and patterns; this objective is set in light of realising the benefits of object-orientation and reuse.

We believe that the formal aspects of the work presented in this thesis can have an impact on goal (7). In doing so, further contributions towards goals (6) and (4) are made.

UML has been standardised by the Object Management Group (OMG) [Gro05] since 1997, in a series of specification documents starting from UML1.1 [OMG97] to UML 1.3 [OMG99] to UML 1.4 [OMG01] to UML1.5 [OMG03b] and most recently with the adoption of the final specification for UML 2.0 [OMG04]. All previous versions include minor updates and refinements, with the exception perhaps of the move from UML1.4 to UML1.5 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 UML2.0 includes a significant update to previous versions and offers interesting perspectives with regard to specification of component-based software. We discuss these features explicitly in the sequel. As a shorthand, we shall refer to previous versions of UML as UML1.x.

First, we briefly outline the core set of UML diagrams and the UML semantic model which is common across UML1.x and UML2.0. Our presentation is based on the specification documents of UML1.x and then on books [BRJ99, RJB99] authored by the people who perceived the move towards a unified approach to software systems visual modelling. The interested reader is also referred to [Si98, EP98, BJR97, PS99] for a
more comprehensive presentation. Notice that some books refer to earlier versions of UML. The most up-to-date specification of the language can be found in [OMG04].

- **Class diagrams** show a set of classes and the relationships between them. A relationship between collaborating classes may be a dependency, a generalisation or an association. Class diagrams are used to model the static design view of a system. They also provide the foundation for two other related diagrams, namely the component diagrams and the deployment diagrams.

- **Object diagrams** show a set of objects, each in a specific state and in a particular relationship to other (collaborator) objects at a point in time. An object diagram is an instance of a class diagram in the sense that it shows a snapshot of the detailed state of a system at a particular point in time. Object diagrams are used to model the static view of a system at a frozen moment in time, and can be useful for modelling complex data structures.

- **Use Case diagrams** show actors and use cases together with their relationships. Use cases define units of functionality or behaviour provided by the system. Actors represent the roles that users of use cases play when interacting with the system and may be humans or automated systems. This inherent sense of interaction with actors outside the system is the primary reason for the adoption of use case diagrams in capturing requirements for the system.

- **Interaction diagrams** show a collection of instances (e.g. objects) that exchange stimuli in order to perform a specific task. An interaction is defined in [OMG03b] as a behavioural specification that comprises a sequence of communications exchanged among a set of instances within a collaboration to accomplish a specific task. Communication takes place via exchange of stimuli. These may include calls to operations, sending signals but also more implicit forms of interaction such as branching conditions. Interaction diagrams are used to model the behavioural aspects of the system and come in two flavours: sequence and collaboration diagrams.

Sequence diagrams emphasise the time ordering of messages exchanged between
participating instances. Collaboration diagrams emphasise the organisation of the participating instances. Note that sequence and collaboration diagrams are semantically equivalent, implying that a diagram in one form can be converted to the other without loss of information. This does not necessarily mean that both diagrams explicitly visualise the same information, but rather indicates that both diagrams share the same underlying model; they both originate from the same information in the UML’s metamodel. The difference is in the way they represent this information.

- **State diagrams** represent state machines while emphasising the flow of control from state to state. A state machine is a graph of states and transitions and it renders the states and responses of an instance to events that it receives (external stimuli). States describe the status condition of an object while transitions specify how these conditions are interrelated. UML state diagrams are a variant of Harel’s statecharts [Har87] and they are used for modelling the dynamic aspects of the system. Dynamic aspects relate to the event-ordered behaviour of the instance in any view of the system’s architecture.

- **Activity diagrams** are a variation of state diagrams intended to model computations and workflows. The main difference is that activity diagrams describe the behaviour of an instance in response to internal processing rather than external stimuli. In contrast to interaction diagrams - collaboration and sequence diagrams - which emphasise the flow of control from object to object, activity diagrams emphasise the flow of control from activity to activity. An activity is an ongoing non-atomic execution within a state machine, in the sense that it may be interrupted and, in general, is considered to take some duration to complete. Activities result in some action (e.g. call to an operation, sending a signal, creating / destroying an object, evaluating an expression etc.). Actions are atomic in the sense that events may occur but the computation within a state is not interrupted. Activity diagrams are used to model dynamic aspects of the system in terms of a workflow or an operation.

- **Component diagrams** show software implementation components, interfaces and
relationships between them. They are essentially class diagrams that define
development-time and run-time physical things such as libraries, executables,
tables, files, etc. Component diagrams are used to model the static implementa-
tion view of the system in terms of configuration management of the files of the
source code or a physical database.

It may be worth pointing out that software components in UML are seen as
software elements that comprise implementation files such as libraries or executa-
bles. Components are not seen as design-level entities and this may hinder the
application of the standard UML to component-based design. We return to this
discussion shortly.

- **Deployment diagrams** contain nodes, on which software implementation compo-
nents reside, together with the communication relationships among them. A
node is a run-time physical object that represents a computational resource with
memory and, possibly, processing capabilities. Deployment diagrams are used
to model the static deployment view of the system in terms of the configura-
tion of processing resource elements and the mapping of software implementation
components onto them.

UML is (semi-)formally defined using a metamodel. The metamodel itself is expressed
using constructs in the UML, thus implying a metacircular interpreter approach; the
language itself is defined in terms of itself. In fact, only a small subset of UML is used
in defining the metamodel.

The UML metamodel is a logical model and not a physical or implementation model.
As such, the metamodel 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 metamodel.

The UML metamodel [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)

• 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 metaclasses that show strong cohesion with each other and loose coupling with metaclasses 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 subpackages, 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 subsystems, which serve as grouping units for UML model elements.

The three top-level packages of the UML metamodel, together with their subpackages, are shown in Figure 2.4 found in 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 metamodel using the metamodel itself. This is counterbalanced though by the fact that a fair enough 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
Figure 2.4: The package structure of the UML metamodel
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 the same thing.

Further, the UML metamodel seems to focus on defining the relationship between groups of UML concepts, in the form of packages and their subpackages, rather than 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. (We point out such cases in various points of the thesis.)

So far we have outlined the core set of concepts and constructs that underlie UML. 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, its application to component-based software design as such is not fairly obvious.

Apart from some generic problems associated with UML due to a lack of a precise (i.e. formal) semantics, the main reason the language can not be applied to component-based software design in a straightforward manner is that the UML concept of a component is tightly related to its ‘physical’ nature. In particular, UML1.x has a standard component concept which represents low level software units that exist at run-time. Design abstractions such as provided and required interfaces and the dependencies between them are not issues of concern for components.

Components in UML1.x appear in the component and deployment diagrams where they are seen as executable files that contain the implementation of concrete classes (see Figure 2.5).

Component diagrams, which are essentially class diagrams, are used to model the static implementation view of the system as they define run-time physical things such as libraries, executables, files, etc. Deployment diagrams, also a variation of class diagrams,
2.2. Graphical Descriptive Approaches

Figure 2.5: A software component in UML1.x notation

go one step further, though sticking with the same component concept, and map software implementation components onto actual processing resources. Hence, the concept of a component as a physical piece of implementation is suitable for the purpose of these diagrams. However, a pure component-based approach to software engineering that realises the promise for reusable components, rather than recycling code, emphasises component interface specifications and requires a clear separation of specification aspects of design from implementation choices.

Note that component and deployment diagrams employ the same component concept in UML2.0 [OMG04] but this more recent update to the specification of the language includes a notion of a component at the specification level and provides some additional notation to accommodate it. We discuss this shortly.

Early approaches to overcome the problem with the component concept in UML1.x, prior to the advent of UML2.0, within the context of specifying dependencies between provided and required interfaces, such as [KF02], and later [MS03], suggested the use of the UML lollipop notation for provided interfaces and the dependency relationship, borrowed from standard UML class diagrams, for required interfaces. Components were rendered as stereotyped classes using the «comp spec» stereotype. This is depicted in the component specification architecture of Figure 2.5 where component A provides two interfaces i1 and i2 and requires a service from component B through its required interface i3.

This draws upon concepts of the component-based software development process prescribed by Cheesman and Daniels in [CD01]. The authors define a pragmatic extension of UML1.x in an attempt to capture important component concepts at the specification
Figure 2.6: Component specification architecture using UML1.x notation

level. The graphical notation they use exploits the stereotype extensibility mechanism defined in UML and is conformant with the UML1.x notation (in fact, they apply UML1.3 [OMG99]).

In particular, the stereotype <<comp spec>> is used to denote component specifications rather than standard classes, in the class diagrams used to model the specification artifacts of the system. Interfaces are also introduced as stereotyped classes using the <<interface type>> stereotype. This is because such modelling elements refer not only to the syntactic interface description but also include the information model of the interface. The information model in [CD01] captures a view of the state model of the component in terms of provided operations, attributes associations and concrete classes and is understood as defining the usage contract of the interface. The information model of [CD01] takes the form of one or more classes having at least one composition relationship with the interface it refers to.

The stereotype <<interface type>> is used in [CD01] instead of the standard UML stereotype <<interface>> because the latter would not allow the state models to be associated with interfaces in this way. A standard UML class stereotyped by <<interface>> can be understood as a potential realisation of an <<interface type>> stereotyped class. Similarly, a useful way to think about the relationship between a
2.2. Graphical Descriptive Approaches

<<comp spec>> and the standard UML modelling element for a component, is to consider a UML component as a realisation of a <<comp spec>>, in the usual 'implementation realises specification' sense. These ideas are illustrated in the diagram of Figure 2.7 which depicts the specification of a component $A$ with a provided interface $I_1$, together with their implementation artifacts.

![Diagram of UML extensions for component specification](image)

**Figure 2.7: UML1.x extensions for component specification**

Furthermore, the approach of [CD01] for describing component contracts can be seen as an extension to the design by contract approach [Mey92], tailored to software components. The provides / requires dependencies between components are described by constraints. Component contracts are represented partially declaratively (using OCL as specified in [OMG99]) and partially operationally (using UML collaboration diagrams). In fact, OCL is used for describing pre- and postconditions of interface operations while collaboration diagrams are used for capturing component interactions. For instance, the usage contract in [CD01] is described using OCL.

However, OCL (even as defined in [OMG03b]) has been designed as an annotation notation rather than a full specification technique for expressing behavioural constraints on the UML model elements. Consequently, it lacks the appropriate depth of expressiveness to describe component contracts precisely. For example, the behaviour of a component receiving a request on one interface and reacting to it by requesting a service on another interface cannot be expressed in OCL at the moment, though it
might be possible in future versions of OCL (see [OMG03a] for the latest specification; [KW00, BKS02] for approaches towards enhancing the expressive power of OCL; [CKF04] for expressing liveness properties with OCL and [KFA05] for adding the notion of time to OCL).

On the other hand, OCL stands for Object Constraint Language and is thus intuitively meant to be applied to objects rather than a more coarse-grained concept such as components. Nevertheless, the need for a similar notation for components has lead [KF02] to adopt a Catalysis [DW99] like notation for a declarative description of component contracts. The same, or similar, motivation can be found in [KFOLO00, LO99] for describing interacting frameworks. In this context, frameworks can be understood as components with particular care for the context in which they are expected to be deployed.

The adoption of the final specification of UML2.0 [OMG04] constitutes a significant update to previous versions of UML. Some of these changes offer interesting perspectives for the specification and analysis of component-based software.

In an attempt to support the notion of components throughout the modelling lifecycle UML2.0 adopts a component concept at the specification level on top of the implementation focus of UML1.x. A component in UML2.0 (see chapter 8 in [OMG04]) 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 interfaces. 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 UML2.0 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 UML2.0 semantics hints towards a formal contract but this is not provided or prescribed in the specification.

Graphically, a component is represented in UML2.0 as a Classifier (e.g. 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 smaller rectangles protruding from its left-hand side just as in component and deployment diagrams of UML1.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 2.8 depicts a component in UML2.0 notation with two provided interfaces p1, p2 and three required interfaces r1, r2, r3.

![Component Diagram](image)

**Figure 2.8: A software component using UML2.0 notation**

A component can be embedded into any environment (or system) that satisfies the constraints 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. Component interfaces allow for the specification of both structural (e.g. attributes, association ends) and behavioural features (e.g. 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 UML2.0 by using dependencies between component interfaces. This is typically done in *structure diagrams*. These diagrams show components and connections between them in terms of matching 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.)
Chapter 2. Approaches to Modelling Component-Based Software

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 2.9 shows a component specification architecture where an assembly connector is used to connect the required interface of component A to the provided interface of component B and another to connect the required interface of component B to the provided interface of component C.

![Figure 2.9: A structure diagram for connecting components in UML2.0](image)

Furthermore, UML 2.0 introduces the *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 component-based software 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 handling.

Figure 2.10 shows a component A\(B\) which relies on the collaboration between the contained components A and B for fulfilling its component contract in terms of its provided interface \(p_1\) and its required interfaces \(r_1\) and \(r_2\).

![Diagram](image)

Figure 2.10: A UML2.0 composite structure diagram for component A\(B\)

Finally, another change in UML2.0 has to do with state diagrams. This is a minor update to previous versions but can prove useful in describing the behaviour of components at their interfaces.

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 UML2.0, which
in addition introduces protocol state machines for expressing the usage protocol of a part of the system.

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 behavioural state machines, 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 component 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. UML2.0 explicitly considers a notion of conformance of PSMs 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 of 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 specialisation of the latter.

Interestingly, one of the constraints specified on PSMs (see [OMG04], p. 584) states that if two interfaces are connected then the PSM of the required interface must be conformant to the PSM of the provided interface. 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.
2.2.3 Scenario-based descriptions

Sequence diagrams in UML are used to describe interactions between entities of the system. An interaction in UML, or scenario more 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 often used in design where the precise inter-process communication must be set up according to specified protocols.

UML sequence diagrams, Message Sequence Charts (MSCs) [IT96], and Live Sequence Charts [DH01a] are popular for describing interactions. These scenario description languages (SDL) are becoming more expressive and despite their increasing sophistication they are the mainstay of industrial software specifications (e.g., see [BBJ+02]).

Graphically, a sequence diagram has two dimensions. The vertical dimension represents time. Time progresses top-down the diagram. The horizontal dimension represents instances participating in the interaction and these can be understood as objects or processes or components (or, more generally, as a Classifier in UML dialect). An individual instance participating in the interaction is represented by a dashed vertical line, called lifeline. A lifeline usually has a rectangle on top containing its instance name (identifier). Communication between instances appearing in a sequence diagram is performed by sending and receiving messages. A message may reflect occurrence of an event such as an operation call, sending/receiving a signal, raising an exception. Messages are denoted by horizontal arrows drawn from the source lifeline to the target lifeline.

The semantics of a sequence diagram defined with any of UML, MSC, LSC notations is given in terms of a partial order on the events appearing in the interaction described in the diagram. However, the partial order induced by a sequence diagram 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. Such inconsistencies are often referred to as race conditions. For example, $b$ and $c$ as experienced by component
$B$ in the sequence diagram of Figure 2.11 are in a race condition.

![Figure 2.11: A UML sequence diagram with a race condition](image)

The vertical dimension of the diagram suggests that $a$ happens first, then $b$, then $c$, and finally $d$. However, this is not necessarily the case. This is because the event of sending $b$ is ordered to occur before the receiving of $c$ (the ordering is imposed along the lifeline of $B$), but the sending of $b$ and the sending of $c$ are not ordered (they belong to different lifelines). Therefore, $c$ may be sent before $b$ or even at the same time as $b$ (since $A$ who is responsible for sending $c$ after $a$ does not know whether $B$ has already sent $b$ or not). As a result, there is no way to ensure that $b$ will occur before $c$.

Typically, MSCs are used to specify scenarios as sequences of messages exchanged between objects or processes. In fact, UML sequence diagrams are a variant of MSCs while LSCs are an extension of MSCs for liveness. MSCs have also been proposed for component-based software design in [FK01] together with a formal underpinning based on Büchi automata for formal verification. This work is also concerned with the issue of component composition in terms of the corresponding MSCs. However, only sequential composition is considered and the parallel construct in a MSC is not covered.

The wide acceptance of MSCs lead to their standardisation by the telecommunications industry with the ITU standard (see [IT96] and [IT00] for the latest specification). Yet, their interpretation can be ambiguous; for instance, does a MSC describe how the systems will always behave or does it give a possible behaviour of the system? According to the ITU standard MSCs only do the latter. But then, virtually nothing can be said
in MSCs about what the systems will do when the described scenario actually occurs.

LSCs [DH01a, HM03] were proposed as an extension to MSCs that addresses this issue as they can explicitly distinguish between mandatory, possible and forbidden behaviour. This is done by adding liveness to individual parts of the chart (or diagram). Within a chart, the live elements, termed hot describe mandatory behaviour - things that must occur. When used properly, hot elements can describe forbidden behaviour, i.e. disallowed sequences of interactions. Other elements, termed cold, describe possible behaviour - things that may occur. They can be used to capture conditional behaviour and various forms of iteration. In terms of notation, all hot elements are indicated in solid lines/boxes/hexagons while cold elements in dashed lines/boxes/hexagons.

LSCs have one of the following two modes: existential and universal. An existential chart has a dashed borderline and it describes behaviour that must be observed in some execution of the system. A universal chart, see Figure 2.12, is made of two superimposed parts: the upper one, annotated with a dashed borderline hexagon, is the prechart and the lower one, annotated with a solid borderline rectangle is the main chart. A universal LSC induces an action-reaction relationship between its prechart and main chart. Whenever the behaviour contained in the prechart is observed, the system must exhibit the behaviour described in the main chart.

![Diagram of a universal LSC](image)

Figure 2.12: A universal LSC

LSCs are extensively used in the play-in/play-out approach [Har01] to system develop-
ment, which is supported by an automated tool, the Play-Engine [HM03, HM04]. The play-in part concerns specifying scenario-based behaviours via a user-friendly interface (GUI). The designer first builds the GUI of the system under development and then 'plays' the incoming events on the GUI (by clicking buttons, sending messages and so on). Again through the GUI (e.g. by right-clicks), the designer then describes the desired reactions of the system and the conditions that may or must hold. During this process, the Play-Engine automatically constructs the corresponding LSCs which capture the cause-effect relationships of the system. The play-out part is concerned with executing the behaviour modelled in the universal LSCs while the scenarios were played in. When playing out, the designer restricts her/himself to external environment actions. During this process the Play-Engine keeps track of actions and causes other actions and events to occur as prescribed in the corresponding LSCs.

In addition to capturing the intended behaviour, the play-in/play-out approach is meant to test and validate it as well. This assumes that during play-out the designer operates the system freely rather than restricts to the exact scenarios that were played in. The approach offers new perspectives with regard to software development, especially for reactive systems, even though it leaves the designer with the onerous task of building an 'appropriate' GUI that allows for a complete set of scenarios to be played in.

The 'live' aspect of LSCs, reflected by their temperature notion, makes it possible to express liveness constraints, that is, that eventually something happens enforcing progress of the instances along their lifeline. The expressiveness of LSCs in this respect is therefore similar to temporal logic. On this note, [KF04a] describes how behaviour specified in LSCs can be expressed in the branching time distributed temporal logic MDTL [KF00b, KFO0a]. This is done within the context of defining an extension of LSCs for describing agent interactions. The extension mainly consists of introducing decision subcharts for expressing nondeterministic choice in an LSC, but also concerns a more complex message sending mechanism (in terms of explicit and arbitrary message sending) to accommodate agent/role interactions.

[KF04a] describes how formulae in MDTL can be derived from (enriched) LSCs. MDTL is a two-level logic initially developed for the specification of object-oriented modules.
The top level logic, called *communication* logic, is used to describe inter-object specification and can be understood as a way to model an observer of the interaction as a whole. The low level logic, called *home* logic, is used to describe intra-object behaviour and can capture local state invariants and interaction constraints from an instance viewpoint. The home logic is a first-order temporal logic with an additional concurrency operator while the communication logic is defined over a predicate logic. Their combination adds a distribution flavour to the temporal nature of MDTL. This makes it possible to express the behaviour of LSC hot elements in MDTL formulae, for example, that whenever a message is sent it must eventually be received.

We have seen that the inception of LSCs was mainly motivated by the MSCs limitation in expressing necessity; interaction patterns that must rather than may occur. UML2.0 sequence diagrams are also limited in expressing necessity. Drawing upon concepts of LSCs, the operator assert was introduced (see [OMG04], pages 412, 442) to model mandatory behaviour. However, the UML2.0 specification does not clearly define whether this operator enforces a sequence of events to happen or they are simply expected to happen. Further, as pointed out in [KF04b], even if the former case was the intended one, it still only solves the problem at the interaction level (either of the whole interaction or of the sub-interaction within assert), but not at the level of a single message or progress along a lifeline.

To address this limitation, [CKF04] proposes to decorate sequence diagrams with liveness constraints which are expressed in an OCL template defined in [BKS02]. The template essentially consists of an *after*: clause used to express a trigger and an *eventually*: clause used to express that a certain condition will eventually hold. This makes it possible to express necessity in sequence diagrams as a liveness property - whenever the *after*: clause holds, at some point in the future the *eventually*: clause must hold.

Sequence diagrams in UML2.0 [OMG04] have been considerably revised in relation to those of UML1.x and have been extended to include features from MSCs and, to a lesser extent, from LSCs. As a result, UML2.0 sequence diagrams are more expressive and fundamentally better structured. One of the major changes has to do with the introduction of sub-interactions called *interaction fragments* which can be combined using
interaction operators. Interaction fragments comprise one or more operands (compartments) depending on the corresponding interaction operator. There are several possible operators which can be used in an interaction fragment for describing various kinds of behaviours. For example, there is a seq operator for describing sequential behaviour, alt for alternative behavior, par for parallel behaviour, neg for forbidden behaviour, loop for iteration, and so on. Hence, the semantics of the resulting sub-interaction depends upon the operator used and is described informally in the UML2.0 superstructure specification document (see [OMG04], ch. 14).

The lack of precise behavioural semantics in the specification for the 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 (see [OMG04], pages 403, 410) hints towards considering all possible interleavings in the resulting sequences of events. According to this semantics, it is not possible to differentiate between the behaviour described in the sequence diagrams of Figure 2.13(i) and (ii).

Figure 2.13: The parallel construct in UML2.0 and a possible interpretation

Following an interleaving interpretation, the sequences of event occurrences in the diagram v1, pictured in Figure 2.13(i), says that either b1 occurs before b2, or, b2 occurs before b1. This is precisely the behaviour described in Figure 2.13(ii) even though the

\[\text{A more detailed description of the basic interaction operators in UML2.0 and their semantics is given in Chapter 3, Section 3.2.1, where UML2.0 sequence diagrams are used for constraining the behaviour of a component.}\]
alt operator is used this time. This may not be an issue when all we are interested in is that both \( b1 \) and \( b2 \) have occurred at the end of the interaction - this implies an implicit synchronisation point at the end of the diagram \( vl \). If, however, we want to include the case that \( b2 \) and \( b1 \) occur at exactly the same time then, diagram \( v2 \) no longer describes the intended behaviour of instance \( B \). The situation gets even more complicated if we were to insist on \( b1 \) and \( b2 \) occurring at exactly the same time (e.g. consider a tuner component of a TV blocking the audio output on the speakers and the video output on the screen before changing the frequency).

A parallel construct is also used in MSCs for describing parallel behaviour. This is denoted by the keyword \( PAR \), just like in Figure 2.13(i), or by a coregion, similarly to LSCs. The semantics of the parallel construct given in [ITO0] simply says that events appearing within the construct are unordered. This means that the partial order along the lifeline collapses within the scope of the parallel construct.

LSCs also include a construct for parallel behaviour, the so-called coregions which correspond to regions in the diagram where the events are unordered. They are annotated with dotted vertical lines around the events corresponding to the sending and the receiving of the concurrent messages. The symbolic transition system presented in [DH01a] implements an interleaving interpretation of coregions. LSCs were subsequently given a semantics in terms of timed Büchi automata in [KW01] and this allows for a true-concurrent interpretation of coregions. This approach is further discussed below.

Our, almost periodic, reference to the semantic issues of the scenario-based description languages highlights the fact that, although useful for informal documentation and triggering discussion during design, these languages cannot be relied upon for rigorous analysis and formal verification. Therefore, scenario specifications need to be translated into other, more formal, notations. This can determine a precise interpretation if the target notation has a well-defined semantics.

[KW01] translate scenarios into a SDL specification using timed Büchi automata [AD92]. This approach considers concurrent events as occurring in either order including simultaneity. Other than addressing all possible relationships between unordered events (ap-
pearing in a coregion construct in LSCs), the need to include simultaneity in [KW01] arises when considering shared condition valuations, but also zero delay messages as imposed by the target verification tool for STATEMATE [HN96] designs, namely the STATEMATE Verification Environment (STVE) [BDW00]. The translation of an LSC into a timed Büchi automaton involves adding timing information (time intervals) to LSCs using a notation similar to that for specifying delays in MSCs [AHP96]. Hence, in this approach concurrency is not considered as an explicit structural property of the corresponding automata, but rather is expressed in terms of their timing properties.

[KF04b] translate scenarios described in UML2.0 sequence diagrams into labelled event structures [WN95]. In addition to the main features of UML2.0 sequence diagrams, this work addresses sequence diagrams combined with an OCL liveness template as proposed in [CKF04] discussed earlier. It builds on the event structures model [NPW81, Win88] which allows to describe distributed computations as event occurrences together with order-theoretic relations between them. Additional information from a sequence diagram is attached to the formal model in the form of labels, e.g. messages, state invariants, condition valuations, etc.

Further, [KF04b] shows how an interaction described in a sequence diagram can be expressed as a collection of formulae in the distributed temporal logic Mdtl. This is done using the two main features of this logic, namely distribution (by means of a communication and a home logic) and concurrency. In addition, OCL liveness constraints attached to the diagram are translated into the home/communication logic depending on whether they refer to a global or local constraint.

Mdtl is given a semantics in terms of labelled event structures (see [KF00b, KP00a] and thus the notion of concurrency considered is based on that of the event structures model where events are understood as being concurrent if they do not precede each other and are not mutually exclusive. Hence, the approach of [KF04a] to formalising interactions described in UML2.0 sequence diagrams considers events appearing within a parallel construct as being unordered but does not model simultaneity explicitly.

[WS00] translate scenarios into statecharts. It is not all that clear though how these statecharts are to be composed to provide the overall system behaviour. We have more
to say about this approach when we discuss it in the context of combining multiple scenarios which is an issue we describe next.

Another problem with regard to scenario-based specifications has to do with managing multiple scenarios. Each scenario is a partial story which needs to be combined with other scenarios to provide a more complete description of system behaviour. As far as component-based design is concerned, all scenarios a particular component participates in need to be considered in order to obtain a complete description of its behaviour.

_Interaction overview diagrams_ (IODs) have been introduced in UML2.0 as a means of combining various scenarios (interactions, in UML dialect). IODs show how multiple interactions are considered, through a variant of activity diagrams, in a way that promotes the overview of the flow of control between different interactions (scenarios). Each node in an IOD is either a whole sequence diagram or a reference to an existing sequence diagram (this is called InteractionUse in UML dialect). Alternative interaction fragments in a sequence diagram are represented in IODs by a Decision node and a corresponding Merge node. Parallel interaction fragments are represented by a Fork node and a corresponding Join node. The flow of control in IODs determines the acceptable ordering between interactions (scenarios).

The ITU standard [IT00] also provides syntactical constructs for combining scenarios expressed in MSCs. These are inline expressions (similar to InteractionUse in UML2.0 IODs) and high-level MSCs (hMSCs) (similar to IODs in UML2.0). hMSCs are directed graphs, see Figure 2.14, where each node is either a MSC or a hMSC and edges of the graph indicate the acceptable ordering of scenarios.

![Figure 2.14: A high-level MSC](image-url)
The idea, other than bringing together the various scenario specifications and showing how these relate, is that hMSCs allow for reuse of scenarios and introduce sequencing, loops and alternatives between MSCs.

hMSCs, just like IODs, are useful for informal documentation but cannot be relied upon for more rigorous approaches to analysis and verification of systems.

A more rigorous approach to synthesis of multiple scenarios, expressed using MSCs, is that of [UKM03]. The authors generate a FSP (Finite State Processes) specification from a given set of MSCs and then use the LTSA (Labelled Transition System Analyser) [MK99] to generate a LTS model of each component. These are then combined to obtain a model of the complete system. The step from FSP to LTS is performed automatically by the LTSA tool. The step from MSCs to FSP involves state labels and hMSCs.

State labels are used to identify component states that, although appearing on different MSCs, i.e. different scenarios, actually refer to the same component state. This allows the component to switch between those MSCs. The MSC specifications are annotated with state labels for the start and the end of each MSC, and the hMSC determines a relation between 'starts' and 'ends' of different MSCs. Each lifeline is then split into sub-lifelines such that all sub-lifelines start and end on labelled states and have no other labelled state in between. Next, sub-lifelines from different MSCs are matched on the basis of common labelled states. Each sub-lifeline is translated into a local FSP process by using the label of its start state as the process name and its sequence of events as the process behaviour. The final state of the process is another local process corresponding to another sub-lifeline that can be followed after the one being dealt with.

As a small note here, it is not entirely clear in the above approach, at least as described in [UKM03], how the state labels are inserted in the first place (and how naming consistency is guaranteed across stakeholders), how agreement is reached on the hMSC (as this diagram is a driver for the step from MSCs to FSP) and why instances not participating in a particular scenario sometimes appear in the corresponding MSC. Finally, the LTS semantics of FSP impose an interleaving interpretation of concurrency. Note that concurrency is expressed in MSCs using the parallel construct whose semantics only says that messages appearing in the construct are unordered - without excluding
2.2. Graphical Descriptive Approaches

[AEY03] consider MSCs for component-based design and propose a language-theoretic framework in which scenarios in a MSC specification are translated into a deterministic FSP process that has the same language as the projection of the MSC specification on the alphabet of the component. The complete FSP model is then fed into LTSA, just like in [UKM03] discussed above. Further examination of the execution traces that are output by LTSA may reveal implied scenarios. hMSCs in [AEY03] are also represented by LTS where transitions are labelled by basic MSCs (these are MSCs appearing in hMSCs). The language accepted by the LTS of the hMSC is the set of maximal basic MSCs in the original MSC specification. The mechanics of this composition of basic MSCs in the original MSC specification, in terms of the corresponding MSCs, is not simply the product (or shuffling) of the corresponding transition systems but is not described in detail in [AEY03].

Another approach to combining scenarios is described in [WS00]. The authors start with a set of MSCs and assume some pre-existing domain knowledge about the participating components that has been specified in OCL. Then, they specify a set of state variables for a component together with pre- and post-conditions for its interactions again expressed using OCL. A valuation of state variables, termed state vector, is then assigned to every state of a MSC using the pre- and post-condition information on messages appearing in the interaction. These valuations can be used to (a) detect loops within a lifeline, (b) identify similar states between lifelines of the same component (and so on different MSCs) and, (c) introduce hierarchy in the generated statecharts. In fact, a LTS is synthesised from the combination of the MSCs and the OCL specifications.

As a small comment here, it could be argued that the use of OCL, which is part of UML, to describe domain knowledge about components of the system is advantageous - in the sense of promoting standards. The term 'domain knowledge' seems to capture all knowledge and expertise required to properly select a set of state variables that can provide all necessary information (e.g. loops, reference to same state) throughout the MSC specification. Hence, the approach assumes that the scenario providers (or stakeholders or component designers or real world practitioners) can and are willing to
express the domain knowledge in OCL and always care to describe system behaviour from its initial state.

2.3 Formal Approaches

In the previous section we were concerned with graphical descriptive techniques for modelling software systems, and in particular component-based systems. Diagrammatic notations are useful for visualising and communicating ideas but cannot support more rigorous approaches to software engineering, unless they are equipped with a precise formal semantics. We have described instances of work that is aimed at providing a formal underpinning to diagrams, especially with respect to scenario-based graphical notations.

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 set of interacting components.

General-purpose formal methods such as Z [Spi92] or VDM [Jon90] could be useful in specifying the behaviour of component-based software. 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 objects, 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 component-based software design.

In addition, software components are increasingly expected to operate in a distributed and concurrent setting [KFPRS04]. This makes stronger demands in terms of component interactions and parallel behaviour. 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], event structures [NPW81, Win88], asynchronous transition systems [Shi85, Bed88], π-calculus
2.3. Formal Approaches

[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 Z and CSP has been proposed in [RS99] or, more recently, a combination of Object-Z and CSP in [DB02].

One combination of formal methods that stands out is that of combining CSP with the formal development method B [Abr96]. 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.

The idea is to use B for the state-based description of the system, capitalising on the fact that B machines are particularly suited for modelling parallel activity, and use CSP for the event-based description, capitalising on the fact that CSP process descriptions provide a natural way of modelling sequential activity by ordering event occurrences. In brief, to marry the two, each event in the CSP description is associated with an operation in a B machine. Below we give a brief account of the combination beyond this high-level concept.

[But00] proposes the csp2B tool for converting CSP descriptions of system behaviour into standard B specifications. Hence, the combination of CSP and B in this case is achieved by compiling the CSP into a pure B representation that can be subsequently analysed by a standard B tool. The translation of a CSP process to a pure B specification involves techniques such as inserting implicit states, adding state variables and introducing auxiliary functions in addressing the various CSP operators. The CSP support in csp2B does not include internal choice and event hiding while interleaving is restricted to multiple instances of an indexed process running in parallel (where the instances do not interact with each other in any way).

This approach to combining CSP and B provides the B specification with a more natural way of expressing ordering constraints. However, it could be argued that this comes at the expense of a particular interpretation of the construct for parallel activity in B. The multiple substitution operator [||] of B, which in principle can express parallel activity such as simultaneous change of state, simultaneous valuation of conditions and
so on (see [Abr96], p.243, 308-312), is reduced to CSP's parallel composition operator '||', which is a synchronisation on common events rather than parallelism. Also, the CSP interleaving operator '|||' is restricted to multiple instances of the same process, as mentioned before. The corresponding B machine introduces synchronisation points to address '|||' of CSP and the use of B's '||' is reduced to imposing these synchronisations between interleaved instances of the process.

[ST04] takes a view of the combination of B and CSP which takes the form of a B machine and a controller written as a sequential CSP process description. Given a B machine \(M\) and a CSP controller \(P\), the combination, the so-called controlled component, is the parallel combination of the CSP controller and its B machine, denoted by \(P || M\). The parallel combination is achieved by giving B machines a semantics based on Morgan's CSP-style semantics for event systems, as done in [Tre00].

The idea behind this form of combining B and CSP is that controlled components consist of a B machine controlled by CSP controllers, under a particular architecture [ST02] in which interaction across the system can only occur between the CSP controllers. This work builds on the concept of the control loop invariant [Tre00] on the state of the B machine for verifying individual controlled components. The model checker FDR [Ltd97] can be used for analysing CSP processes, and thus CSP controllers, individually and in combination. The authors show in [ST04] how the results obtained on the CSP part of the specification (CSP controllers) can be lifted to the combined specification (controlled components).

The CSP semantics of the combination allow B machines to be treated as CSP processes. As such, they are combined in [ST04] using the parallel composition operator '||' of CSP. This approach offers interesting compositional results with regard to preservation of properties, such as divergence freedom and deadlock-freedom, under composition of controlled components. On the other hand, it can be argued that this view of a component, as a pure CSP process, is somewhat restrictive. For instance, it does not allow for concurrent events on distinct component interfaces. (Recall that a CSP controller is a (recursive) sequential CSP process.)

A slightly different approach to the combination of CSP and B is proposed in [BL05]
where the CSP and the B specifications are composed in parallel. The B operations must synchronise with channel events of the CSP process description having the same name as the B operation. Channel events of the CSP process that are not associated with an operation in B can occur independently while operations in B which are not associated with a channel event in CSP are prohibited. The authors describe an extension to the ProB [LB03] model checker for B that supports checking the proposed combination of CSP and B. The tool is targeted at checking consistency between B and CSP specifications, that is, no B pre-conditions are violated and the B specification satisfies the trace properties of the CSP specification. The CSP support in the extended ProB tool, unlike csp2B, includes arbitrary combinations of CSP operators.

In the remainder of this chapter, we discuss formal approaches to the analysis and specification of component-based software. In particular, we describe representatives of approaches originating from various branches of mathematics such as algebra, logic, regular expressions, transition systems and automata.

### 2.3.1 Algebraic approaches

[Bro00, Bro95] present an algebraic model for components. The basic idea behind this approach to formal specification of reactive component-based systems 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 component-based systems in a fashion similar to the use of algebraic specification techniques for data structures and information flow.

The algebraic approach of [Bro00, Bro95] advocates the use of *streams* to describe communications on the channels of a software 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 $M^\omega$. Hence, $M^\omega = M^* \cup M^\infty$ where $M^*$ denotes the finite sequences over $M$, including the empty sequence, and $M^\infty$ denotes the infinite
sequences over $M$.

Concatenation and prefix ordering operations are introduced on streams. The set of streams $M^\omega$ equipped with the prefix ordering relation is complete in the sense that every directed set $S \subseteq M^\omega$ has a least upper bound. Recall (e.g. [DP90]) that a directed set $X$ is defined as a non-empty 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. Furthermore, [Bro95] defines functions for selecting the first element of a stream and removing the first element of a stream, providing the stream is non-empty.

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 $O$ 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

\[ f : (I \to M^\omega) \to (O \to M^\omega) \]

that maps every input history onto an 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 $B$ (true or false) on the stream processing functions of $C$. This defines a set of deterministic behaviours $Q$ as follows.

\[ Q : ((I \to M^\omega) \to (O \to M^\omega)) \to B \]

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 $x \in (I \rightarrow M^a)$ 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 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 functions 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 the 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 $C_1||C_2$ and the channels of the composite are given by

$$Out(C_1||C_2) = Out(C_1) \cup Out(C_2) \quad \text{and} \quad In(C_1||C_2) = In(C_1) \cup In(C_2)$$

where $Out(C_i)$, $In(C_i)$, 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. The parallel composition of two components $C_1$ and $C_2$ is depicted in Figure 2.15 found in [Bro00].

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 \( x \in In(C) \) and \( y \in Out(C) \) the feedback operator \( \mu_y^C \) describes the feedback of the stream output from channel \( y \) to the input channel \( x \). It is defined by \( In(\mu_y^C) = In(C) \setminus \{x\} \) and \( Out(\mu_y^C) = Out(C) \), where \( In(C) \setminus \{x\} \) 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 component-based software 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.
2.3. Formal Approaches

It might be worth noting here that this builds on earlier work on introducing time to the model through *timed streams* in [BRS00]. 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 $M^*$. This allows to isolate the stream containing the elements of the first $t$ time intervals.

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 (e.g. 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 of one component happen 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.

Another approach to formalising the behaviour of components at their interfaces is that of [PBJ98, PV02, PVB01, AP03]. The authors propose a formal framework for describing the ordering of events on component interfaces based on the use of behaviour protocols [PV02, PVB01] which take a form similar to regular expressions (see e.g. [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 [vdBL91]. The approach of [PV02, PVB01] 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 software 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 \(< F, A >\) 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 subcomponents 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 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 subcomponent (this is termed delegation) and between a required
2.3. Formal Approaches

interface of the nested component and a required interface of a subcomponent (termed subsuming).

In terms of a formal description, a component $c$ in [PV02, PVB01] is considered within its environment (a collection of other components) and is assumed to have a set of connections $V$ (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. words $\in A_c^*$. For an event $a$ in $A_c$, a request and a response associated with $a$ is denoted by $a\uparrow$ and $a\downarrow$, respectively. Issuing a request (method call) is denoted by $!a\uparrow$ while accepting a response by $?a\downarrow$. (If a request and a response occur inside a nested component (between its subcomponents) then the corresponding events are prefixed by $\tau$, e.g. $\tau a$.)

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 (bounded) by a behaviour protocol which is essentially a type of regular expression that generates a set of traces over $A_c$.

Being a regular-like expression, a behaviour protocol is constructed using classic operators used in regular expressions such as sequencing ($;$), alternatives ($+$) and repetition ($^*$), but may also employ additional operators such as interleaving/shuffling ($|$), restriction ($/$) and composition ($\Pi_X$). More details can be found in [PV02]. Here, we find it sufficient to discuss the composition operator in a bit 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 components communicating via connected interfaces. (These should be a provided of one component and a required interface of the another, though not explicitly stated in [PV02] or [PV02] or [AP03].) 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 interleaving/shuffling 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 re 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 II_\times is applied to protocols, the resulting composite 
protocol P_1 II_\times 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 composi-
tion operator, where the originator of complementary events cannot be determined. 
In subsequent work [AP03], they attempt to address aspects of the problem by way of 
introducing error events and erroneous traces.

However, the notion of composition (under II_\times) considered in this approach seems 
to infer a rather strong synchronisation assumption to the model since it yields valid 
traces only when a request event is immediately followed by the response event. Notice 
that this is stronger than the usual 'handshake communication' which says that the 
send event on one end of the communication channel is (immediately) experienced as 
a receive event on the other end. Composition in this approach makes the further 
assumption that the response to the send event (request) is generated immediately 
afterwards (which is not necessary in handshake communication).

The potential pitfalls are experienced for instance when investigating the use of UML2.0 
Protocol State Machines (PSMs) 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, that 
generate the communication language of a behaviour protocol. Not surprisingly, the 
variation points mostly relate 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 component interfaces, which seems to be unsubstantiated given that, 
in principle, a PSM may comprise sub-PSMs in orthogonal regions (see [OMG04], p. 
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 (on distinct interfaces of the same component, not on connected interfaces of different components). Thus, events that happen concurrently on two required interfaces, perhaps in response to some event received on a provided interface, cannot be modelled.

As a final 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 relationships to new events (due to the different configuration) need to be specified. In this respect, a notation for the language that expresses events on each interface separately would appear to be more suitable.

### 2.3.2 Logic-based approaches

Another approach to formal modelling of component-based software has been proposed in [KF02]. This work draws upon standard algebraic concepts but is also blended with the logical framework developed in [KF00a, KF00b] which plays a central role in specifying the behaviour of components. A component in [KF02] is understood as a unit of software that has a number of provided and required interfaces for communication with its environment and other components. The focus in this approach is on specifying component contracts, the dependencies between provided and required services of components.

A component contract in [KF02] consists of a *usage* contract and a *realisation* contract. The usage contract defines the details of a contract with clients of the component in
terms of the operations it provides, including their signatures, their effects and when these effects can be guaranteed to hold. The realisation contract defines how the provided interfaces are related to required interfaces and may contain constraints on the implementation of an operation (given through corresponding component interactions).

This notion of a component contract picks up on ideas from [CD01], where a component is understood as a single class and its instances are complex objects, but here a component is instead understood as a collection of interacting object classes.

The specific vocabulary considered to be relevant for the description of the structure of a component is given in [KF02] using order-sorted signatures. An order-sorted data signature is defined as a triple \( \Sigma_D = (S_D, \Omega_D, \leq_D) \) where \( S_D \) is a finite set of data sorts, \( \Omega_D \) is an \( S_D \times S_D \)-indexed family of sets of data operations and \( \leq_D \subseteq S_D \times S_D \) is an ordering relation that turns \( \Sigma_D \) into a partially ordered set. Moreover, the partial order \( \leq_D \) is monotone (in that it satisfies the condition that if \( o \in \Omega_{D_{1,\ldots,n}} \times \Omega_{D_1,\ldots,D_n} \) and \( s \leq_D r_i \), for \( i = 1..n \), then \( s \leq_D r \)) which allows to deal with partial functions and express inheritance and polymorphism in object-oriented languages. Recall that a component in this approach is understood as a collection of objects. For this reason, object and component sorts, denoted by \( S_O \) and \( S_C \), are considered in addition to data sorts. A partial order on object sorts reflects an inheritance relation (each class is equipped with an object sort). A partial order on component sorts denotes subcomponent relationships. Note that only sorts of the same kind can be related by the partial order \( \leq \subseteq S \times S \) where \( S = S_D \cup S_O \cup S_C \).

The above are summarised in the notion of a kernel signature which is an order-sorted signature \( \Sigma = (S, \Omega, \leq) \) where \( S = S_D \cup S_O \cup S_C \) and \( \Omega \) is an \( S^* \times S \)-indexed family of sets of operation symbols such that \( \Omega_D \subseteq \Omega \) and includes extra operations for object instances, attributes, actions and component instances.

The use of order-sorted signatures in describing the structure of a component in [KF02] allows to infer well-known results from category theory. In particular, it allows to define morphisms between order-sorted signatures in such a way that the signatures and morphisms form a category. A morphism is considered in [KF02] as a function mapping symbols of one signature to symbols of the other. This is exploited in defining import
and export signatures as inclusion morphisms over kernel signatures which map the component sort of one kernel signature to a component sort of the other. Components are combined on the basis of matching import and export signatures. This leads to the definition of a component signature $\Theta = (\Sigma, \text{Imp}, \text{Exp})$ where $\Sigma$ is a kernel signature and $\text{Imp}, \text{Exp}$ are finite sets of import and export signatures over $\Sigma$.

In order to capture the dependencies between provided and required interfaces and hence describe component contracts, [KF02] uses the distributed temporal logic $\text{MDTL}$, which was discussed before. A component with a signature $\Theta$ is associated with a component logic $\text{MDTL}_\Theta$ which in turn associates to each of its subcomponents a local logic $\text{MDTL}_{\Theta_m}$, where $m$ is the component term of the subcomponent.

The local logic of a component is split into a home logic $H_m$ and a communication logic $C_m$. This dichotomy underlines the distributed nature of the logical framework. The home logic is used to describe intra-component communication and internal component properties while the communication logic is used to capture the knowledge about other components and describe inter-component communication.

The home logic $H_m$ is a first-order temporal logic with an additional concurrency operator. Formulae in the home logic can be obtained by applying successively the connectives $\neg$ and $\Rightarrow$, the temporal operators $U$ (until) and $S$ (since), the concurrency operator $\wedge$ and the $\forall$ quantifier to atomic formulae. An atomic formula can be the logical constant $true$; the predicate for comparison $\theta$; the predicate $\Phi$ (enabled) and the predicate $\Theta$ (occurring), which are applied to operations and essentially reflect pre- and postconditions of operations. The logical constant $false$ and the well-known connectives of propositional calculus such as $\land, \lor, \leftrightarrow$, the quantifier $\exists$ as well as the other temporal operators like $\text{next } X$, $\text{sometimes } / \text{always in the future } F, G$ etc. can be derived from the basic $\text{MDTL}$ operators. Details of such derivations can be found in [KF00a].

The communication logic $C_m$ is used to capture the knowledge $m$ has about other components, gained through communication. There are three possible statements in $C_m$ which are used to express the occurrence of a synchronous communication action, the occurrence of a send and a receive asynchronous communication action respectively.

A component in this approach is formally defined as a component description $CD =$
\((\Theta, Ax)\) where \(\Theta\) is a component signature and \(Ax\) is a set of axioms in its corresponding component logic, i.e. \(Ax \subseteq \text{MDTL}_\Theta\).

In terms of component specification, the export signatures correspond to provided interfaces and the import signatures correspond to the required interfaces of a component. Possible constraints reflecting the dependencies on other components (through required interfaces) can be expressed in the communication logic of the component. Axioms reflecting pre- and post-conditions on interface operations can be expressed in the component’s home logic.

\text{MDTL} is given a semantics using an order-theoretic model of event structures [NPW81, Win88]. This has been mentioned before but we find it appropriate to discuss the semantic foundations of this approach to component specification in a bit more detail.

In [Win88], distributed computations are viewed as event occurrences together with a binary relation for expressing time ordering, in terms of causal dependency. This relation is taken to be a partial order among event occurrences and gives rise to elementary event occurrences. A second relation can be considered on the resulting partially ordered set of events, namely conflict, which is symmetric and irreflexive, and is used to express nondeterminism. This leads to the definition of prime event structures. Based on these two relations, causal dependency and conflict, it is possible to derive a further relation to denote concurrency. Concurrent events are events which are neither related by causal dependency nor by conflict.

In giving a semantics to \text{MDTL}, [KF00a, KF00b] considers a restriction on prime event structures that deals with finiteness and results in the definition of discrete prime event structures. The restriction reflects the fact that a system’s computations always have a starting point and excludes infinite descending chains of event occurrences. This allows a further relation to be derived, termed immediate causality which characterises immediate predecessors or successors of event occurrences.

In order to link discrete prime event structures to \text{MDTL} and establish their use as a denotational semantics for the logic, a labelling function is attached to the set of events. In effect, the labelling function is a (total) function from the set of events to a set of labels for actions and maps each event to an action symbol or a set of action symbols.
2.3. Formal Approaches

Event structures equipped with the labelling function are called labelled prime event structures in [KF00a, KFOb] and provide the semantics of the logic used in [KF02] for formalising component contracts.

The notion of concurrency considered in this approach is that of the order-theoretic model of event structures, often referred to as true-concurrency. Events are considered to be concurrent if they do not precede each other and are not in conflict. This notion of concurrency is the one found across partial order models of computation. It might be worth noting that it does not exclude simultaneity but does not model it explicitly either. Having said that, this notion of concurrency seems to be more suitable for expressing parallel activity in a component setting.

2.3.3 Automata-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 adapters for the purpose of reusing components in different configurations.

This approach takes a black-box view of a component 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 interface of the component while the required gates define its required interfaces.

Component 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 specialisations of such FSMs are used to model the observable behaviour of a component in terms of change of state due to method or operation calls. FSMs without actions, denoted by P-FSM, are used to model the behaviour of a provided interface. FSMs without events on transitions, denoted by M-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-FSMop to the target state of op in the initial P-FSM. A detailed 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. Figure 2.16 is an anonymised version of an example used to demonstrate the construction in [SR00]. Note the use of the UML notation c/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 adapters, 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 method/operation calls to two provided interfaces. This comes down to merging the sequences of operations of the 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 possi-
ble 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-FSM 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 work of [SR02a, SR00, Ren00, Ren99] is 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 UML2.0 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 to a single provided interface does not allow for parallel behaviour of the component. This limits a component to servicing requests sequentially only when concurrency could be realised (e.g. through replication of objects, code 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 distinct 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 P-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 in 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 \textit{i}. The postfixes are determined as paths from \(t(s_C, s_P)\) 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-condition),
and vice versa, in defining the component contract. In contrast to classic 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., provided) 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 parametric 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 automata-based approach to the formal description of component behaviour
is that of [dH01c, dH01b]. [dH01c] 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
[dH01b] 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 [dH01b] consists of: i) a finite set of states; this includes an initial state, ii) input actions; these can be understood as events (e.g. operation calls and their return values) on the receiving end of communication channels, iii) output actions; these can be understood as events (e.g. 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; this defines a step between states via some action. It might worth pointing out that the definition of an interface automaton, as given in [dH01b], does not include any deterministic condition, effectively allowing multiple target states for a single transition.

A component in [dH01b] 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 discuss composition of interface automata in a bit 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 [dHO1b] 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 in [SR02a], [dHO1b] 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 check 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 [dHOlb] 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 component 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 of 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 [dHOlb] which is essentially synchronisation 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 only one automaton, and thus do not cater for the full range of independent actions, as discussed before.

2.4 Concluding note

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 software components. In this chapter we have reviewed approaches to the specification and analysis of component-based software ranging from pure formal specification techniques to solely diagrammatic notations.

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 graphical 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. To be more precise, it is not exactly error prone, especially since there is no universal definition of what constitutes a good or even correct diagram, but the meaning of a diagram is often ambiguous.

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, UML2.0 [OMG04] 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 UML2.0 (see ch 8, ch 15 in [OMG04]), such a formalisation is not laid out in its specification document.

More generally, graphical descriptive techniques do not provide component designers with a standard way of expressing behaviour at component interfaces. Sequence diagrams have been introduced for this purpose, but the lack of a precise behavioural semantics allows, for example, different sequences of executions to be derived depending upon the interpretation of the parallel construct, or implicit scenarios due to race conditions to go unnoticed. As a result, whether using UML2.0 [OMG04] or MSC [IT96] or LSC [DH01a] notation, when the scenarios described in a sequence diagram are executed certain anomalies could come to view.

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 descriptive techniques, including UML, do not commit to a specific formal semantics, allows for a number of formal interpretations to be attached. We have discussed such work in Section 2.2.

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 towards 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 underlie 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 UML2.0 which have multiple provided interfaces.

Further, 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, nondeterminism and simultaneity, and reveal pathological behaviour that arises through the interaction of such phenomena.

The review of existing formal approaches to component-based software design 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 common between components (on bound provided and required interfaces from each) while allowing potentially concurrent execution of all other events.

Further difficulty arises when considering concurrency at the level of individual components. This is the case of events co-occurring on distinct interfaces of a component. 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 within the formalism.

The interleaving interpretation of parallel behaviour, as well as corresponding parallel constructs in UML2.0 sequence diagrams, MSCs, and LSCs, in existing formalisms for components, with the exception of [KF04b, KF02] which consider true-concurrency, enforces 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 arbitrary 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 event structures [NPW81, Win88], is needed for expressing concurrency at the level of individual components. Further, in a component setting it would be appropriate to model explicitly the case that 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.

A formal model that encompasses the above, and also provides a formal underpinning to UML diagrams concerned with specification aspects, would allow for a formal framework which uses a significant subset of UML, e.g. components, sequence diagrams, (protocol) state machines, composite structure diagrams, state diagrams, for the specification and analysis of component-based software.

The usual UML notation such as class diagrams, object diagrams, state diagrams and so on, could then be used to model the implementation aspects of components (assuming an object-oriented approach), thus making use of the full power of UML and the popularity it enjoys among software practitioners today.
Chapter 3

A Formal Language for Components

In the previous chapter, we have seen common approaches to component-based development and graphical representations of components. In this chapter, we give a formal definition of a component that captures both its static characteristics (structure) and dynamics (behaviour). Initially, the view we take of a component is as liberal as possible in that we consider all possible behaviours on its interfaces. However, in component-based design we are interested in modelling the intended behaviour of a component only. For this purpose, we describe a formal specification technique that uses UML 2.0 for constraining the behaviour of a component.

UML 2.0 sequence diagrams provide a graphical notation, with a rich set of features, which is concerned with the interactions between entities of the system. Our interest in sequence diagrams lies with specifying the behaviour of a component at its interfaces. We treat a subset of their features, those deemed useful for capturing component interactions, and describe a formal construction that unfolds a sequence diagram into component vectors. In doing so, we give a concrete formal semantics for these features using a vector language. We also add two features that are not covered in UML 2.0 sequence diagrams: the concept of a location, borrowed from LSCs, and a new construct which allows to specify the simultaneous observation of several events.
3.1 Formalisation of a Component

In this section we formalise the concept of a component within our framework. Below, we state the assumptions that encapsulate the approach taken to the formalisation of a component.

We have seen in Chapter 2 that the main driver behind the component-based way of working in software development is the reuse of previously constructed components. A component is considered as an autonomous unit within a system, which provides services to other components and requires services from other components of the system. It has different provided interfaces through which subsets of its offered services are made available to other components, and different required interfaces through which it requests services from other components.

This view of a component is reflected in the way components are understood and pictured in UML2.0 [OMG04] and the Koala component model [vOvdLKM00], which underlie current approaches to industrial software design. It allows for a modular architecture that facilitates replacement and adaptation of components in different reuse contexts, by connecting them together via provided and required interfaces. The challenge lies with capturing the dependencies of a component in such a way that it can be treated as independently as possible. This is where our formal description of a component, in terms of its interactions via its interfaces, comes into play.

A component in our approach may have multiple provided interfaces through which it makes its services available and multiple required interfaces through which it issues requests to other components in order to deliver its offered services. A provided interface may be related to more than one required interfaces. For instance, in response to a call to an operation on one of its provided interfaces the component may make calls to operations (on interfaces provided by other components) through its required interfaces.

The description of the behaviour of a component at its interfaces in our approach evolves around the primitive notion of an event which is something that occurs on a component interface. An event is defined in the ODC [Oxf96] as something that
3.1. Formalisation of a Component

happens in a computer system. In our study of components we are concerned with one additional aspect. An event in this thesis is understood as something that happens which one can choose to regard as indivisible. This does not imply that we forbid events from having some internal detailed structure - and such structure could well be analysed at a lower level of abstraction. At that more detailed level of abstraction though, what was originally an event is no longer a single event, but several. Therefore, this more abstract view of events we take does not seriously affect our explanations.

Furthermore, by viewing events as having no detailed structure, we may expect events to be localised, in the sense that an event occurs in a small area and within a small period of time. This means that it is appropriate to consider their occurrence as instantaneous.

To sum up, in viewing the events of a component-based system, rather than being focused on the detailed structure of an event we are more interested in how an event influences other events; how the occurrence of an event causally depends on the (previous) occurrences of other events. This view has been proposed in [Lam78] and is characteristic of the understanding of events in others (e.g. time ordering in [Win88], causal dependency in [KFLO+00, KFOOb], causality in [Bro00, Bro04]). It is manifested in the behavioural presentation model [Shi88] that is used in the event-oriented description of component behaviour, given in Chapter 6.

It may be instructive to note that this view of events is also consistent with the way events are considered in UML. An event in the specification document of UML 2.0 [OMG04] is defined as, and we quote, "the specification of a significant occurrence that has a location in time and space and can cause the execution of an associated behaviour. In the context of state diagrams, an event is an occurrence that can trigger a transition". The specification aspect gives us the freedom to work at a high level of abstraction in which it is still possible to capture the conceptual causal ordering between occurrences of events. The second part of the definition shall prove useful when we consider automata for components, in Chapter 7, and especially with regard to their transition structure.

The treatment of events as instantaneous allows to consider that the occurrence of any
event is at a single precise time; within a time slice. This implies that occurrences of events in the system do not blur into one another, and can be judged for simultaneity. Our formal model of a component would be classified as branching-time in the categorisation of concurrency models given in [NSW94]. Time as such is not modelled in our formal framework but it is understood to progress with reference to a global conceptual clock and at the same rate for all components in the system. This notion of time is referred to as Newtonian time in [Sch00].

The fact that components are put together by connecting (wiring in UML dialect) a required interface of one to a provided interface of the other implies that an event issued by the required interface must be one that is understood by the provided interface, and can be serviced at the time it takes place. For example, an operation on the provided interface might be called, which is available but not ready at the time of the call because some other operation must have been called first. So there is a specific sequence of events a component can engage in through one of its interfaces. Further, a component is connected to different components through different interfaces. Some services provided to or required from other components are independent of each other and so can be performed concurrently.

This is reflected in our approach by considering that events occur sequentially on a single interface while they may occur concurrently on distinct interfaces of a component. Hence, events associated with a particular interface only occur one at a time whereas events from different interfaces may occur at the same time. This allows to consider concurrency at the level of individual components so long as the events in question engage distinct interfaces of the component.

Finally, an additional issue has to do with the mode of communication. In synchronous communication the sender is blocked until the recipient responds. In asynchronous communication the sender is free to engage in other activities until a response is received. Our formal model for components can cope with the synchronous case in a straightforward manner. It can also cope with asynchronous communication, but under the condition that the interface(s) on which the response is expected are not involved in the subsequent behaviour (until the asynchronous call has been completed). This condition
3.1. Formalisation of a Component

is required to ensure that an unfortunate situation of simultaneous events on a single interface is always avoided.

Further, both zero-delay and delayed communication can be considered within our framework. In the case of delayed communication however, we make the further assumption that events sent are experienced (received) in the same order at the receiving end. We will have more to say about this when we consider the composition of components in Chapter 5.

These assumptions underlie the component notion in our formal framework for the specification and analysis of components. A component in our approach is considered in terms of a component signature, which serves to identify the component, and a component language, which can be used to model its behaviour. We describe each in the following sections.

3.1.1 Component Signature

A component at the specification level can be seen as a software entity that provides services to other components or the environment and, possibly, requires services (which can be viewed as its 'pre-condition') from other components in order to deliver those promised (viewed as its 'post-condition'). The offered services are made available via a set of provided interfaces while the reciprocal obligations are to be satisfied via a set of required interfaces.

We have seen that graphically a component is often rendered as a square box with a number of provided and required interfaces in UML2.0 [OMG04] or with a number of input and output ports in Koala [vOvdLKM00]. Figure 3.1 depicts a component using the notation of UML2.0 which was described in Chapter 2.

![Figure 3.1: Graphical representation of a component](image-url)
To formalise such a picture, and in light of the contractual use of components [Mey92, Szy97, DW99, HC01, Rau02], the static semantics of a component is captured in terms of two disjoint sets of interfaces. Those the component requires and those the component provides. Furthermore, the static semantics specifies for each interface the set of operations it supports. Let $I$ be the set of names for interfaces and $Op$ be the set of operations associated with interfaces in $I$, both sets remaining fixed throughout this study.

**Definition 3.1.1.** We define a component signature to be a tuple $\Sigma = (P, R, \beta)$ where

- $P \subseteq I$ is a set of provided interfaces
- $R \subseteq I$ is a set of required interfaces
- $\beta : P \cup R \rightarrow \wp(\text{Op})$; hence, $\beta(i)$ is the set of operations associated with interface $i$ of the component

and we require that $P \cap R = \emptyset$. Define $I_\Sigma = P \cup R$ and $Op_\Sigma = \bigcup_{i \in I_\Sigma} \beta(i)$.

By $\wp(\text{Op})$ we denote the powerset of $\text{Op}$, i.e. the set of all subsets of $\text{Op}$.

These sets and this function comprise the static characteristics of a software component. They serve to identify a component. Hence, the signature of a component can be considered as its static specification. For simplicity, and at the level of abstraction that our component model shall be considered, we refer to events that may occur on an interface as operation calls. However, these could be understood in more general terms as input actions (on provided interfaces), used to model calls to operations/procedures/methods as well as the return locations or responses of such calls, and as output actions (on required interfaces) which are used to model operation/procedure/method calls made by the component and exceptions that may arise during execution. In this sense, the notion of a component signature resembles the alphabet structure of interface automata [dH01b].

For instance, in an object-oriented approach, a component may comprise several classes which might call operations/methods of other components, react to events generated
by other components and the environment, accept calls from other components, or throw exceptions that can be handled by other components or the environment. These are the kinds of events that would occur on the interfaces of a component within our component model. Hence, they may be, largely, understood as operation calls made at and by the component while offering services to other components and the environment.

**Example 3.1.1.** Consider the component $A$ of Figure 3.1. It has two provided interfaces $i_1$ and $i_2$, hence $P_A = \{i_1, i_2\}$. It has one required interface $i_3$, hence $R_A = \{i_3\}$. Thus, its set of interfaces is given by $I_{DA} = \{i_1, i_2, i_3\}$. Also, $P_A \cap R_A = \emptyset$. Assume that the component $A$ accepts calls to operations $a_1, a_2, a_3, a_4$ on its provided interface $i_1$, and $b_1, b_2$ on $i_2$ and can make calls to operations $c_1, c_2, c_3$ through its required interface $i_3$. This information is given in terms of the function $\beta$ as defined in Definition 3.1.1. In this case we have,

\[
\begin{align*}
\beta_A(i_1) &= \{a_1, a_2, a_3, a_4\} \\
\beta_A(i_2) &= \{b_1, b_2\} \\
\beta_A(i_3) &= \{c_1, c_2, c_3\}
\end{align*}
\]

The set of all operations associated with the component is given by $Op_{DA}$ as defined in Definition 3.1.1. In this case we have

\[
Op_{DA} = \bigcup_{i \in I_{DA}} \beta_A(i) = \{a_1, a_2, a_3, a_4, b_1, b_2, c_1, c_2, c_3\}
\]

The signature of component $A$ is given by $\Sigma_A = (P_A, R_A, \beta_A)$ where

- $P_A = \{i_1, i_2\}$
- $R_A = \{i_3\}$
- $\beta_A : P_A \cup R_A \to \mathcal{P}(Op_{DA})$ is given by $\beta_A(i_1)$, $\beta_A(i_2)$ and $\beta_A(i_3)$ as above.

It can be seen from the example that in addition to the information conveyed by the graphical representation of the component in Figure 3.1, the corresponding component signature specifies the set of operations associated with each of its interfaces.
3.1.2 Component Language

The formal description of a software component includes a dynamic part, which describes the behaviour of the component in question. In addition to the static specification of a component we are also interested in how it will be behave on its interfaces. This is the dynamic specification and it is captured in our framework using component vectors.

Initially, the view we take of the dynamic characteristics of a component is as unrestricted as possible. In any behaviour of the system, each interface of the component will experience a sequence of events (e.g. operation calls made at or from that interface). The idea is that the behaviour of the component as a whole can be described by assigning such a sequence to each interface of the component. For this purpose, we introduce the notion of component vectors in our model.

**Definition 3.1.2.** Suppose that $\Sigma$ is a component signature. We define $V_\Sigma$ to be the set of all functions $\gamma : I_\Sigma \rightarrow Op_\Sigma^*$ such that for each $i \in I_\Sigma, \gamma(i) \in \beta(i)^*$. We shall refer to elements of $V_\Sigma$ as component vectors.

By $\beta(i)^*$ we denote the set of finite sequences over $\beta(i)$. A function $\gamma$ of the definition is a tuple of sequences since it maps each interface of the component to a finite sequence of events (e.g. operation calls) that have occurred on that interface. It has one coordinate for each interface of the component. During execution, an interface $i$ of the component will experience a number of different sequences of events formed over its corresponding set $\beta(i)$. For example, consider a prefix of a given sequence. These different sequences are captured by different $\gamma$. Thus, each different function $\gamma$ associates at least one interface with a sequence of events that does not appear in any other $\gamma$. The set of all such possible functions $\gamma$ comprises the set $V_\Sigma$ and we refer to them as component vectors.

Mathematically, the set $V_\Sigma$ is the cartesian product of the sets $\beta(i)^*$, for each $i \in I_\Sigma$. Component vectors are $n$-tuples of sequences where each coordinate corresponds to an interface (hence $n$ is the number of component interfaces) and contains a finite sequence of events (e.g. operation calls, exceptions) that may occur on that interface. The set
3.1. Formalisation of a Component

\( V_\Sigma \) of all such vectors formed over a component signature \( \Sigma \) comprises the set of all possible behaviours for the component associated with \( \Sigma \).

The set of component vectors \( V_\Sigma \) is central to our formalism so let us take a closer look on what these vectors look like and how they are formed.

For a component with \( n \) interfaces we have component vectors of the form

\[
(\mathbf{u}(i_1), \mathbf{u}(i_2), ..., \mathbf{u}(i_n)) \in \beta(i_1)^* \times \beta(i_2)^* \times ... \times \beta(i_n)^*
\]

where

\[
\mathbf{u}(i_1) = x_1 \in \beta(i_1)^*
\]
\[
\mathbf{u}(i_2) = x_2 \in \beta(i_2)^*
\]
\[...
\]
\[
\mathbf{u}(i_n) = x_n \in \beta(i_n)^*
\]

It is important to note that, for example, \( x_1 \) is one sequence out of all possible sequences formed over \( \beta(i_1)^* \). That is to say, each coordinate contains a finite sequence of events out of all possible sequences that may be formed over the alphabet (set of events) associated with the corresponding interface.

**Example 3.1.2.** Consider the component \( A \) of Example 3.1.1. We have seen that its signature is given by \( \Sigma_A = (P_A, R_A, \beta_A) \) where \( P_A = \{i_1, i_2\} \), \( R_A = \{i_3\} \) and \( \beta_A \) is given by

\[
\beta_A(i_1) = \{a_1, a_2, a_3, a_4\}
\]
\[
\beta_A(i_2) = \{b_1, b_2\}
\]
\[
\beta_A(i_3) = \{c_1, c_2, c_3\}
\]

The component vectors associated with component \( A \) are formed over its corresponding signature \( \Sigma_A \). They will have the form of triples where each coordinate corresponds to an interface of component \( A \). By Definition 3.1.2, we have \( \mathbf{u}(i_1) \in \beta_A(i_1)^* \), \( \mathbf{u}(i_2) \in \beta_A(i_2)^* \) and \( \mathbf{u}(i_3) \in \beta_A(i_3)^* \). We may write \( (\mathbf{u}(i_1), \mathbf{u}(i_2), \mathbf{u}(i_3)) \) for \( \mathbf{u} \in V_{\Sigma_A} \), effectively assigning each coordinate of the triples to a particular interface of the component \( A \).

In what follows we use \( A \) to denote the empty sequence.
Chapter 3. A Formal Language for Components

Now, \((a_1a_2, b_1, c_1c_2)\) is a component vector in \(V_{\Sigma_A}\) with \(v(i_1) = a_1a_2 \in \beta_A(i_1)^*\), \(v(i_2) = b_1 \in \beta_A(i_2)^*\) and \(v(i_3) = c_1c_2 \in \beta_A(i_3)^*\). It describes behaviour of the component in which a call to operation \(a_1\), followed by \(a_2\), has occurred on interface \(i_1\), a call to operation \(b_1\) has occurred on interface \(i_2\) and a call to operation \(c_1\), followed by \(c_2\), has occurred on interface \(i_3\).

Another component vector is \((a_1a_3a_4, b_1, c_1c_2)\) for which \(v(i_1) = a_1a_3a_4 \in \beta_A(i_1)^*\), i.e. contains a different sequence of operation calls on interface \(i_1\) (as compared to \((a_1a_2, b_1, c_1c_2)\)).

Examples of component vectors for component \(A\) with signature \(\Sigma_A\) include, \(\{(a_1, b_1b_2, \Lambda), (a_2a_4, b_2, c_3), (\Lambda, \Lambda, \Lambda), (a_1a_3a_4, b_1b_2, c_1), (\Lambda, b_2b_1, c_1), (a_1a_2, b_1b_1b_1, c_1)\}\)

Compare with the following vectors which are not component vectors of component \(A\), \(\{(a_1a_2, b_1, a_1c_1), (a_1b_2, b_1, c_2), (a_1a_2a_5, b_2, c_1c_2)\}\)

Each component vector is built up by coordinatewise concatenation, starting with the empty vector. The empty vector assigns the empty sequence, denoted by \(\Lambda\), to each interface of the component. When an event (or, as we will see, a set of simultaneously occurring events) occurs on an interface, it appears on the coordinate that corresponds to that interface. Hence, for a component with, say, three interfaces, we have component vectors of the form \(v = (v_1, v_2, v_3)\) where each coordinate \(v_i\), \(i = 1, \ldots, 3\) is given by

\[
v_i = \begin{cases} v & , \quad v \in \beta(i) \\ \Lambda & , \quad \text{otherwise} \end{cases}
\]

We assume that the \(\beta(i)\), for each \(i \in I_\Sigma\), are necessarily distinct, i.e. operation names are unique on interfaces. This allows us to associate each occurrence of an event (operation call) with a particular interface and can be achieved by simply prefixing each operation by the name of the interface it is defined on (effectively making the sets \(\beta(i)\), each \(i \in I_\Sigma\), distinct). This is a subtle technical difference between the component vectors in our approach and the behaviour vectors in [Shi97], where common events on distinct coordinates are allowed at the cost of a synchronisation constraint similar to the synchronisation parallel operator, \(\|\|\)’, found in CSP [Hoa85].
3.1. Formalisation of a Component

If an event is associated with more than one coordinate of a behaviour vector, then it appears on each of these coordinates when it occurs. On the contrary, we do not allow common events across different coordinates in component vectors. This allows to consider concurrency between (non-common) events on distinct coordinates and thus on different interfaces of the component, as will be shown in Chapter 4 (Section 4.3.2).

We give a formal definition of concatenation on component vectors later (cf Definition 4.1.1). For the moment, it suffices to understand that component vectors are built up by coordinate-wise concatenation of sequences. For example,

\[(x_1, x_2, x_3, y_1, y_2, y_3) = (x_1 y_1, x_2 y_2, x_3 y_3)\]

In fact, they are built up by coordinate-wise concatenation with a specific kind of component vectors, the so-called column vectors (cf Definition 4.2.6) denoting single event occurrences on component interfaces. Column vectors are essentially component vectors such that each of their coordinates is either the empty sequence or a single event (a sequence of length 1). For example, \(e = (a, A, A)\) describes the occurrence of event \(a\) on the interface corresponding to the first coordinate.

We postpone the formal definition of column vectors until Section 4.2 (cf Definition 4.2.6) because the set of component vectors has to meet certain requirements before we can state (with confidence) that what takes a component vector and extends it to another (its successor(s)) is the occurrence of the event(s) appearing in an appropriate column vector.

Note that column vectors may have more than one non-empty coordinate in which case we talk about simultaneous events. These are events that occur at exactly the same time. For example, the column vector \(e = (a, A, c)\) describes the simultaneous occurrence of event \(a\) on the interface corresponding to the first coordinate and event \(c\) on the interface corresponding to the third coordinate. We will have more to say about this when we consider a behavioural presentation for a component in Chapter 6, since we will then be in a better position to give a formal notion of simultaneity.

Example 3.1.3. Consider the component \(A\) of Examples 3.1.1 and 3.1.2. Initially, nothing has happened, so we have the empty vector \((A, A, A)\). Then, suppose that \(a1\) ∈
\[ \beta_A(i) \] happens. It appears on the corresponding (the first) coordinate and we have a component vector \((a_1, \Lambda, \Lambda)\). Next, suppose that \(c_1 \in \beta_A(i)\) happens (denoted by \((\Lambda, \Lambda, c_1)\)). This is recorded in the set \(V_{\Sigma_A}\) in another component vector \((a_1, \Lambda, c_1)\) which is obtained by coordinatewise concatenation as

\[ (a_1, \Lambda, \Lambda)(\Lambda, \Lambda, c_1) = (a_1, \Lambda, c_1) \]

Then suppose that \(a_2 \in \beta_A(i)\) happens (denoted by \((a_2, \Lambda, \Lambda)\)). The fragment of behaviour of the component in which all three operation calls have happened is recorded in yet another component vector \((a_1a_2, \Lambda, c_1)\) which is obtained by coordinatewise concatenation as

\[ (a_1, \Lambda, c_1)(a_2, \Lambda, \Lambda) = (a_1a_2, \Lambda, c_1) \]

and so on for subsequent operation calls on the interfaces of component \(A\).

It is important to note that a component vector provides an ordering between operation calls on a particular interface, but not between different interfaces of the component. For example, the vector \((a_1a_2, \Lambda, c_1)\) tells us that \(a_1\) followed by \(a_2\) have happened on the interface corresponding to the first coordinate and \(c_1\) has happened on the interface corresponding to the third coordinate. In order to infer the ordering between \(a_1, a_2\) and \(c_1\), which we note occurs on a different interface, we need to look at what other vectors have been obtained for the component. In our example, the presence of vector \((a_1, \Lambda, c_1)\), from which \((a_1a_2, \Lambda, c_1)\) was obtained, tells us that \(c_1\) occurs before \(a_2\).

As will be discussed in Chapter 4, this is given by considering the coordinate-wise prefix ordering (cf Definition 4.1.1) between vectors, which determines that \((a_1, \Lambda, c_1)\) describes an earlier part of behaviour than \((a_1a_2, \Lambda, c_1)\) and consequently, \(a_2\) can only happen after \(c_1\) has. This shall become more clear in Chapter 4 where we consider the order theoretic properties of component vectors.

It can be seen from the definition of \(V_{\Sigma}\), and perhaps has been highlighted through the examples, that the set of component vectors \(V_{\Sigma}\) is potentially infinite - it may contain infinitely many component vectors. Note that the sequences appearing in the component vectors themselves though, may contain repetitions but are always finite.

So far, we have defined a component signature (Definition 3.1.1) and the associated set of component vectors (Definition 3.1.2) formed over this signature. The vectors in \(V_{\Sigma}\)
describe all possible behaviours of a component, given its set of interfaces and the set of operations (events) that each interface supports.

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 system functionality, and for this purpose it is crucial to have a description of the expected behaviour of each component before the system is developed, executed and tested as a whole.

Components are developed under (often differing) assumptions about the context in which they will be placed. For instance, a component may be expecting certain signals to arrive consecutively while another is generating them concurrently. Or, more generally, a component may assume that calls to interface operations occur in a specific order and it may behave as desired only when this order is respected. It is the purpose of component-based design to document such assumptions and describe the behaviour of the component in contexts which satisfy those assumptions.

Within our component model this amounts to restricting to an appropriate subset of $V_{\Sigma}$ comprising component vectors that describe intended or permitted behaviour only.

**Definition 3.1.3.** A component $c$ is a pair $(\Sigma, V)$, where

- $\Sigma$ is the signature of $c$,
- $V \subseteq V_{\Sigma}$ is the component language of $c$.

Thus, a component consists of the static structure described by a signature $\Sigma$ together with a 'language' $V$ of component vectors, formed over $\Sigma$. Intuitively, the idea is that the component language indicates possible constraints on the order in which several operations of the component can or should be called.

It might be noteworthy, that there are a number of ways to restrict the $\pi(i)$ to allowed sequences of operation calls. In [BRF03] a finite state machine is attached to each component interface, in which case the allowed sequences are essentially given by the language accepted by the machine. [Mos04] advocates the use of sequence diagrams, LSCs [DH01a] 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. Alternative options could be the use of regular expressions or simply a textual description (use cases) of intended behaviour.

3.2 From UML to the Component Language

In this section we describe a way to restrict to an appropriate subset $V$ of all possible component vectors $V_T$ associated with a component $c = (E, V)$. The approach we advocate uses UML2.0 sequence diagrams (with a non-interleaving semantics for $\text{par}$, a new interaction fragment $\text{sim}$ for simultaneous events, together with a flavour of LSCs) for specifying the allowed sequences of events on component interfaces within the context of a given scenario. The idea is to capture the observed behaviours at each location (graphical position) in a sequence diagram by mapping them onto component vectors. The sequences appearing on the coordinates of the resulting set of component vectors reflect the valid sequences of communication acts (events on component interfaces) described in the sequence diagram. As a result, the obtained set of component vectors captures the intended behaviour of the component and is precisely the component language $V$ of Definition 3.1.3.

3.2.1 Sequence diagrams in UML

Sequence diagrams in UML are used to describe the interactions between entities of the system. An interaction is understood as a unit of behaviour that focuses on the observable exchange of information between the participating entities. Interactions are a common mechanism for modelling systems and are often used in design where the precise inter-process communication must be set up according to specified protocols.

In a nutshell, a sequence diagram displays participating instances as lifelines running down the page and their interactions over time are represented as messages drawn as horizontal arrows between lifelines. Sequence diagrams are useful for showing which
instances communicate with which others and what messages trigger those communications.

Sequence diagrams in UML 2.0 [OMG04] have been considerably revised in relation to those of UML 1.x [OMG03b]. They have been extended to include features from MSCs [IT96] and, to a lesser extent, from LSCs [DH01a] and as a result they are more expressive and fundamentally better structured. In this section, we outline the basic additional features of sequence diagrams in UML 2.0. The presentation has been restricted to those constructs that need to be considered in describing interactions between components. A detailed description of sequence diagrams in UML can be found in [OMG04].

One of the major changes has to do with the introduction of sub-interactions called interaction fragments which can be combined using interaction operators. Interaction fragments may comprise one or more operands (compartments) depending on the corresponding interaction operator. The semantics of the resulting sub-interaction depends upon the operator and is described informally in the UML 2.0 superstructure specification document [OMG04].

Below, we describe briefly the meaning of the interaction operators used in our approach, as given in UML 2.0. A more concrete behavioural semantics (of the subset) of the interaction operators in terms of a vector language is subsequently given in Sections 3.2.3 - 3.2.7. Before we go on to describe the basic interaction operators from UML 2.0 used in our approach, we introduce the concept of a location.

The graphical positions which are associated with event occurrences along the lifeline of an instance in a sequence diagram are of particular significance, especially when the diagram is considered in a formal setting. In the remaining sections of this chapter, we will be concerned with a rigorous approach to extracting the observable behaviours of a component during its correct participation in an interaction described in a sequence diagram. The various event occurrences along the corresponding lifeline is what gives rise to these observable behaviours. In order to formally describe this, we borrow the concept of a location from LSCs [DH01a] which is missing in UML sequence diagrams but is useful semantically. Locations are the points along a lifeline of an instance which
correspond to the occurrence of some event.

In other words, locations are associated with the sending and receiving of messages in the diagram. In our approach, locations are also associated with the start and the end of interaction fragments. We will also find it useful to consider that an instance has at least two locations (as done in [KF05]): an initial location, corresponding to the start of the diagram, and a final location, corresponding to the end of the diagram.

The locations along a particular lifeline and within an interaction operand are ordered top-down. Thus, the order of execution is determined by the partial order induced among these locations. Note that locations within a \textit{sim} interaction fragment and locations from different operands of an \textit{alt} or \textit{par} fragment are not ordered in any way. Not surprisingly, these are the most challenging cases in mapping locations onto component vectors. We will show how such locations are handled in the sequel (Sections 3.2.3 - 3.2.6).

\textbf{Alternatives}

The \textit{alt} interaction operator designates that the interaction fragment represents a choice of behaviour. It may have multiple operands, each offering a different choice, but at most one of them will execute. The operand that does execute must have a guard expression that evaluates to true at this point in the interaction. Effectively, this means that the event occurrences from different operands of an \textit{alt} are mutually exclusive.

There are two variations of the \textit{alt} interaction operator, namely \textit{option} and \textit{break}.

The \textit{opt} interaction operator has only one operand and designates that the interaction fragment represents a choice of behaviour where either the sole operand executes or nothing happens. It is semantically equivalent to an \textit{alt} interaction fragment with two operands where one operand is non-empty and the other is empty.

The \textit{break} interaction operator also has only one operand and designates that the interaction fragment represents a choice of behaviour where either the sole operand executes or the remainder of the sequence diagram (or the enclosing interaction fragment, if any) executes. In other words, it models an alternative sequence of events that is executed instead of the whole of the rest of the diagram.
The interaction fragments resulting from the above operators can be used to model if..then..else or switch constructs.

Parallel

The par interaction operator designates that the interaction fragment represents the parallel execution of the behaviours from the different operands. In short, the resulting par interaction fragment models concurrent processing.

The UML 2.0 specification document does not give a behavioural semantics for the par interaction operator. It hints (p.497, [OMG04]) towards a parallel merge of behaviours where it is perceived that event occurrences from different operands may come in any order in the resulting execution sequence, while events within the same operand retain their order. It also states however that this interleaving semantics is different from a semantics where it is perceived that events may occur at exactly the same time.

It has been recognised that the interleaving perception of parallelism does not cater for the case where events occur at exactly the same time. In other words, only one event is allowed to occur at a time. We feel that such an interpretation is not powerful enough to capture concurrent processing, where the communicating activities run in parallel.

A more natural way to model concurrent processing is to consider that the events may occur at any order or at exactly the same time (simultaneously).

Our interpretation considers unordered events of a par interaction fragment as being observable in any order including simultaneity. This is in line with the view taken in [KW01] which considers simultaneous regions as an interpretation of coregions in LCSs [DH01a], the LSC counterpart of the UML 2.0 par interaction fragment. It is also closer to the thinking found in [KP01b] which gives a true-concurrent interpretation to par, though this work does not consider the simultaneous case explicitly.

In any case, the fact that UML perhaps hints towards an interleaving semantics for par does not affect the actual graphical notation for par and can be understood as a semantic variation point.

In our approach, and throughout this study, we shall be concerned with representing concurrency explicitly and thus depart from the interleaving approach which essentially
Chapter 3. A Formal Language for Components

reduces concurrency to nondeterminism, as postulated for example in [NSW94, WN95]. It is sometimes the case that inconsistencies or pathological behaviours in a component setting arise as a result of an unfortunate interplay between concurrency and nondeterminism. Thus, we opt for a non-interleaving representation of the parallelism between the behaviours from different operands in a \texttt{par} interaction fragment.

We return to this discussion when we consider parallel locations in Section 3.2.5. Further, a formal treatment of concurrency and simultaneity within our overall approach is given in Chapter 4 in terms of the order-theoretic properties of a component language, in Chapter 6 in terms of the associated event-based model, in Chapter 7 in terms of a state-based model and the corresponding automata. We will see that our formal approach is expressive enough to capture nondeterminism, concurrency and simultaneity as distinct phenomena.

Sequencing

The \texttt{seq} interaction operator designates that the interaction fragment represents a weak sequencing between the behaviours of the operands. This implies that:

1. the ordering of event occurrences within each of the operands is maintained
2. event occurrences on different lifelines from different operands may come in any order
3. event occurrences on the same lifeline from different operands are ordered such that an event occurrence of the first operand comes before that of the second operand, and so on.

The resulting \texttt{seq} interaction fragment is the default for sequence diagrams, i.e. all event occurrences that are not in some other interaction fragment are considered to be in \texttt{seq}. For example, all event occurrences in the sequence diagram of Figure 3.3 (Section 3.2.3) are considered as being in a \texttt{seq} interaction fragment and we do not need to denote this explicitly in the diagram.

A notion of strict sequencing is also included in UML 2.0. The interaction operator \texttt{strict} designates that the ordering of event occurrences is maintained throughout the interaction fragment and not only on a single lifeline.
3.2. From UML to the Component Language

Note that, as stated in the UML 2.0 specification document [OMG04], weak sequencing reduces to strict sequencing when the operands apply to a single lifeline (to only one participating instance). In our approach, we are interested in obtaining the 'language part' of a component and therefore will be focusing on a single lifeline (the one corresponding to the component in question). Thus, the distinction between seq and strict is irrelevant in our case.

Also note that weak sequencing reduces to parallel when event occurrences on different lifelines from different operands are involved. It is noteworthy that the interpretation we gave to the par interaction operator is precisely that of point (2) of the seq operator. This offers an alternative way of understanding the semantics of the par operator discussed earlier. The proposed semantics says that what applies to the sequence diagram as a whole (i.e. that event occurrences on different lifelines from different operands are effectively unordered) also applies to a single lifeline within a par operator (i.e. that event occurrences along the same lifeline from different operands in a par are effectively unordered).

Simultaneity

At this point, we add a new concept which allows to specify the simultaneous observation of events. For this purpose, we introduce a new interaction operator in a sequence diagram, the so-called sim.

The sim interaction operator designates that the interaction fragment represents the simultaneous occurrence of all events appearing in its operand (it has a unique operand).

In other words, it models events that have to happen at the same time. This is one of the cases (simultaneity) considered in our interpretation of concurrent events which was discussed earlier. The difference is that the events must occur at exactly the same time in sim, as opposed to may occur at the same time, which is the interpretation of the par fragment. As a result of occurring at the same time, events in a sim interaction fragment stand in exactly the same relationship to all other events in the diagram.

\[\text{We will see how this can be formally described by considering a pre-order relation between causally related events, a conflict relation between mutually exclusive events, and the generated equivalence relation in Chapter 6.}\]
In this sense, a simultaneous region in our approach reminds of the critical region in UML 2.0 (although in a critical fragment the event occurrences are strictly ordered instead of simultaneous) as the behaviours of the region cannot be interleaved by other event occurrences. Effectively, this means that the sim interaction fragment is treated atomically by the enclosing fragment - whether this is some other interaction fragment or the whole diagram.

An example of a sim interaction fragment can be seen in Figure 3.7 (Section 3.2.6). The events e1 and d1 occur at exactly the same time, following the occurrence of event a1. We will see how this situation is handled in mapping locations onto component vectors in Section 3.2.6.

### 3.2.2 Sequence diagrams for components

Before we start mapping locations onto component vectors we explain the context in which we want to use UML sequence diagrams. Our primary motivation is to restrict to an appropriate subset \( V \) of component vectors from \( V_\Sigma \), for a component with signature \( \Sigma \), which describes the intended behaviour of the component.

A sequence diagram describes global system behaviour as well as what is required of individual components for their correct participation in the interaction. In our approach we shall be concerned with the latter aspect since we are interested in obtaining the vector language part of a component (recall Definition 3.1.3) from a sequence diagram. The embedding of individual components into their environment is described in sequence diagrams by the corresponding lifelines (and associated constructs appearing along a lifeline, e.g. interaction fragments). Therefore, we will be focusing on a single lifeline rather than considering all lifelines in the diagram.

Of course, in practice a component will appear in several sequence diagrams. Without loss of generality, we assume that there is only one sequence diagram for each component \( c \). This can be done because Interaction Overview Diagrams in UML 2.0 [OMG04] generate a single sequence diagram. (Similarly, this can be achieved in MSCs through the use of hierarchical MSCs (hMSCs) [IT96].)
We first formalise the interaction described in a sequence diagram. We will find the signature of a UML 2.0 sequence diagram given in [KF05] of great use in what follows, although we will be concerned with a single lifeline of the diagram rather than the diagram as a whole.

**Definition 3.2.1.** Given a sequence diagram and a component $c$ participating in the interaction with signature $\Sigma = (P, R, \beta)$, the component lifeline is formally given by the tuple

$$Cline = (c, Loc, l_0, Op_\Sigma, SE, RE, Path)$$

where

- $c$ is a component identifier
- $Loc$ is a set of locations on the lifeline corresponding to $c$
- $l_0 \in Loc$ is the initial location
- $Op_\Sigma$ is the set of all operations defined on the interfaces of component $c$, i.e.
  $$Op_\Sigma = \bigcup_{i \in I_c} \beta(i)$$
- $SE \subseteq Loc \times \beta(i), i \in R$ is the set of send events of component $c$, experienced at its required interfaces
- $RE \subseteq \beta(i) \times Loc, i \in P$ is the set of receive events of component $c$, experienced at its provided interfaces
- $Path$ is a given set of well-formed path terms for the diagram (we will have more to say about Path when we define the function scope below)

Further, we define two auxiliary functions and associated conditions over $Cline$. The first has to do with the timing of locations along a component lifeline and the second with the scope of a location. To anticipate, these functions will come to fruition when we map each location onto component vectors in subsequent sections (Sections 3.2.3-3.2.8).
We need to determine the relative graphical position of a location along a component lifeline \( Cline \). We do this by defining an injective function

\[
time : \text{Loc} \to \mathbb{N}_0
\]

which associates each location with a natural number according to its position along the lifeline in the diagram, and is assumed given.

There are certain conditions on this function that formulate our intuitive requirements with respect to timing of locations:

1. the initial location has associated time value 0:

\[
time(l_0) = 0, l_0 \in \text{Loc}
\]

2. since \( \time \) is injective, all locations along a component lifeline have necessarily different associated time values:

\[
\forall l_1, l_2 \in \text{Loc} : l_1 \neq l_2 \Rightarrow \time(l_1) \neq \time(l_2)
\]

The function \( \time \) gives a notion of precedence between locations, which allows us to move top-down the diagram along the lifeline in question (cf Definition 3.2.2). We may talk about the previous location \( l' \) of location \( l \) when \( l \) and \( l' \) satisfy \( \time(l') = \time(l) - 1 \). Similarly, for the next location of \( l \).

Note that \( \time \) here does not necessarily mean occurrence time (though within a seq interaction fragment it does), but rather refers to an implicit visual time value according to the layout of the diagram. That is to say, locations with different visual time values may still have the same occurrence times, if they belong to a par or sim interaction fragment (concurrent or simultaneous locations), or mutually exclusive occurrence times, if they belong to an alt interaction fragment (alternative locations). We will see how such locations are treated in the sequel (see Sections 3.2.3-3.2.6).

In the previous section, we saw various interaction fragments that may appear in a sequence diagram and have an effect on the way the corresponding locations are ordered (or unordered in some cases like within a par interaction fragment). Consequently, it
is important to know in which part of the diagram (in what interaction fragment, if any) a location belongs to. Then, we may treat it accordingly as will be described in Sections 3.2.3-3.2.8. Drawing upon concepts introduced in [KF04b] we may talk about the *scope* of a location by defining a function,

\[ \text{scope} : \text{Loc} \rightarrow \text{Path} \]

which associates each location along a component lifeline *Cl ine* with a path term. The path term identifies the various compartments of a sequence diagram. We do not define here the grammar for generating path terms. It suffices to understand that path terms are encoded in such a way that it is possible to distinguish between a location that is:

- inside the main diagram (i.e. does not belong to any interaction fragment). Here a path term has the form \( \alpha.\text{name} \) where \( \alpha \) is a path term, possibly the empty term \( \varepsilon \), and \text{name} is the name of the sequence diagram, given after the keyword \text{sd} on the top left corner of the diagram. For example, the function \text{scope} for the location \( l_1 \) in the sequence diagram of Figure 3.3 returns \( \text{scope}(l_1) = \text{mov} \).
- marking the start of an interaction fragment. Here a path term has the form:
  - \( \alpha.\text{alt}(n) \) for an interaction fragment \text{alt} with \( n \in \mathbb{N}^+ \) operands, where \( \mathbb{N}^+ \) denotes the set of natural numbers excluding zero
  - \( \alpha.\text{par}(n) \) for an interaction fragment \text{par} with \( n \in \mathbb{N}^+ \) operands
  - \( \alpha.\text{sim} \) for an interaction fragment \text{sim} (\text{sim} has no operands, or only one operand)

For example, the scope of location \( l_1 \) in the sequence diagram of Figure 3.4 is given by \( \text{scope}(l_1) = \text{chc}\text{.alt}(2) \). The location \( l_2 \) in the sequence diagram of Figure 3.6 has \( \text{scope}(l_2) = \text{prl}\text{.par}(2) \). Similarly, location \( l_2 \) in the sequence diagram of Figure 3.7 has \( \text{scope}(l_2) = \text{smul}\text{.sim} \).
- inside an operand of an interaction fragment. Here a path term has the form:
  - \( \alpha.\text{alt}(n)\|k \) where \( k = 1..n \) indicates that the location is within the \( k \)-th operand of an alt interaction fragment with \( n \) operands
- \(\alpha \text{par}(n)\|k\) where \(k = 1..n\) indicates that the location is within the \(k\)-th operand of a \text{par} interaction fragment with \(n\) operands

- \(\alpha .\text{sim}\|1\) indicates that the location is within the first (and only) operand of a \text{sim} interaction fragment

(Note that we still need a path term for this case because we will find it useful to be able to distinguish between the locations marking the start/end of a \text{sim} interaction fragment and the locations found within it.)

For example, location \(l_2\) in the sequence diagram of Figure 3.4 has \(\text{scope}(l_2) = \text{chc.\text{alt}}(2)\|1\), since it appears within the 1st operand of the \text{alt} fragment. Similarly, location \(l_4\) in the sequence diagram of Figure 3.6 has \(\text{scope}(l_4) = \text{prl.\text{par}}(2)\|2\).

Location \(l_3\) in the sequence diagram of Figure 3.7 has \(\text{scope}(l_3) = \text{sm\text{lt.\text{sim}}}\|1\). This path term form also allows us to identify a location that is inside an operand of an interaction fragment which in turn is inside an operand of another interaction fragment. For example, location \(l_7\) in the sequence diagram of Figure 3.8 has \(\text{scope}(l_7) = \text{n\text{chc.\text{alt}}}(2)\|2.\text{alt}(2)\|1\).

- marking the end of an interaction fragment. Here a path term has the form:
  - \(\alpha \text{.alt}(n)\) for an interaction fragment \text{alt} with \(n \in \mathcal{N}^+\) operands
  - \(\alpha .\text{par}(n)\) for an interaction fragment \text{par} with \(n \in \mathcal{N}^+\) operands
  - \(\alpha .\text{sim}\) for an interaction fragment \text{sim}

For example, location \(l_6\) in the sequence diagram of Figure 3.4 marks the end of the \text{alt} interaction fragment and has \(\text{scope}(l_6) = \text{chc.\text{alt}}(2)\).

- marking the end of the sequence diagram. Here a path term has the form \(\alpha .\text{name}\).

For example, location \(l_7\) in the sequence diagram of Figure 3.4 marks the end of the diagram and has \(\text{scope}(l_7) = \text{chc}\).

These two additional functions, \text{scope} and \text{time}, together with Definition 3.2.1 is what we need in order to formally capture what is described in a sequence diagram.

It may be worth pointing out that UML sequence diagrams can model both synchronous and asynchronous messages. Since both modes of communication can be supported in
our formal approach, we do not distinguish between synchronous and asynchronous messages in our sequence diagrams in this section. Thus, we use closed non-filled arrowheads on messages, as an alternative graphical notation that does not commit to synchronous or asynchronous messages but denotes a message that can be either. A sequence diagram describing the interactions of a particular component would either have open arrowheads (for asynchronous messages) or closed filled arrowheads (for synchronous messages), as prescribed in UML2.0 [OMG04].

In what follows, we define some auxiliary subsets of the set of locations $Loc$ which will prove useful in providing a construction that gives a component language for each component participating in the interaction described in a sequence diagram.

We may talk about locations marking the start of an interaction fragment by defining a subset of $Loc$ for each of the three kinds of interaction fragment we are considering,

$$\text{Loc}_{\text{start,alt}} = \{ l \in \text{Loc} \mid \text{scope}(l) = \alpha.\text{alt}(n) \}$$

$$\text{Loc}_{\text{start,par}} = \{ l \in \text{Loc} \mid \text{scope}(l) = \alpha.\text{par}(n) \}$$

$$\text{Loc}_{\text{start,sim}} = \{ l \in \text{Loc} \mid \text{scope}(l) = \alpha.\text{sim} \}$$

Dually, and by making use of the path term marking the end locations of an interaction fragment we may define,

$$\text{Loc}_{\text{end,alt}} = \{ l \in \text{Loc} \mid \text{scope}(l) = \overline{\alpha.\text{alt}(n)} \}$$

$$\text{Loc}_{\text{end,par}} = \{ l \in \text{Loc} \mid \text{scope}(l) = \overline{\alpha.\text{par}(n)} \}$$

$$\text{Loc}_{\text{end,sim}} = \{ l \in \text{Loc} \mid \text{scope}(l) = \overline{\alpha.\text{sim}} \}$$

We may also isolate the locations that are associated with an event occurrence (whether being at the receiving end or at the sending end is irrelevant at this stage). Such locations comprise the set
\[ Loc_e = \{ l \in \text{Loc} \mid e \in \text{Op}_e : (l,e) \in \text{SE} \lor (e,l) \in \text{RE} \} \]

We may take this a bit further, and define locations appearing within an interaction fragment. Since such locations will not be marking the start or end of the fragment they will be necessarily associated with some event occurrence and thus will be subsets of \( Loc_e \). So we may determine locations belonging to a \textit{par} interaction fragment with \( n \) operands as follows.

\[ Loc_{par} = \{ l \in Loc_e \mid \text{scope}(l) = \alpha.\text{par}(n) \mid k, k = 1..n \} \]

Similarly, for locations belonging to an \textit{alt} interaction fragment we have,

\[ Loc_{alt} = \{ l \in Loc_e \mid \text{scope}(l) = \alpha.\text{alt}(n) \mid k, k = 1..n \} \]

Similarly, for locations belonging to a \textit{sim} interaction fragment we have,

\[ Loc_{sim} = \{ l \in Loc_e \mid \text{scope}(l) = \alpha.\text{sim}(n) \mid 1 \} \]

Now, the set of locations involved in some event occurrence, but not belonging to any interaction fragment (this essentially comes down to saying that these belong to a \textit{seq} interaction fragment), may be obtained as follows.

\[ Loc_{seq} = \{ l \in Loc_e \mid l \notin Loc_{par} \cup Loc_{sim} \cup Loc_{alt} \} \]

In the remaining sections, we describe how interactions specified in a sequence diagram are translated into component languages. This involves unfolding the diagram into component vectors. The basic idea is to map all locations along the lifeline corresponding to the component in question onto (a set of) component vectors. This is done by introducing a function \textit{vec.map} from the set of locations to the powerset of component vectors \( V_E \). The component vectors resulting from this mapping capture the observable behaviours at each point in the diagram.

Each component vector provides a \textit{snapshot} of what events have already occurred on the component's interfaces. Starting from the top of the diagram and subsequently
moving downwards, we obtain snapshots of the complete behaviour pattern the component is intended to follow. The corresponding component vectors form a subset of all component vectors in $V_\Sigma$, where $\Sigma$ is the component signature, and describe the intended behaviour of the component.

For readability, we introduce the function $\text{vec-map}$ incrementally.

### 3.2.3 Sequential locations

We start by considering how we can move down a sequence diagram, from one location to the next along the component lifeline in question, whilst mapping each location to (a set of) component vectors.

The component vectors associated with each location are obtained from the vectors associated with the immediately preceding location, by concatenating the event (if any) corresponding to the location being considered with the sequence of events appearing on the appropriate coordinate of the component vectors of the immediately preceding location. The initial location of a component lifeline is mapped onto the empty vector $\Delta\Sigma$.

There are some cases however, in which this central idea does not apply. In particular, the end location of a $\text{par}$ interaction fragment as well as the end location of an $\text{alt}$ interaction fragment need to be treated differently. This is because we have to take into account the various execution sequences that may arise when encountering these interaction fragments. Furthermore, the first location of each operand of an $\text{alt}$ or $\text{par}$ interaction fragment has to be considered in relation to the start location of the $\text{alt}$ or $\text{par}$ fragment rather than its immediately preceding location. This is due to the fact that the visual time does not correspond to the occurrence time for the locations of these interaction fragments. We will see exactly how these special cases are addressed in the following sections.

At this stage it suffices to understand that the aforementioned cases are excluded from the basic definition for moving down the sequence diagram. In other words, we want to exclude the following locations:
• the end location of an \texttt{alt} interaction fragment with \emph{n} operands,

\[ X_1 = \{ l \in \text{Loc} | \text{scope}(l) = \alpha.\text{alt}(n) \} \]

• the end location of a \texttt{par} interaction fragment with \emph{n} operands,

\[ X_2 = \{ l \in \text{Loc} | \text{scope}(l) = \alpha.\text{par}(n) \} \]

• the first location of each operand in an \texttt{alt} interaction fragment with \emph{n} operands,

\[
Y_1 = \{ l_k \in \text{Loc}, 1 \leq k \leq n \mid \text{scope}(l_k) = \alpha.\text{alt}(n) \} \cap \text{time}(l_k) = \text{time}(l'_k) + 1 \land \text{scope}(l'_k) = \alpha.\text{alt}(n) \}
\]

In further explanation of the notation, \( l_k \) is the first location of the \( k \)-th operand if the previous location, \( l'_k \), is a location of the \((k-1)\) operand or the start location of the fragment.

• the first location of each operand in a \texttt{par} interaction fragment with \emph{n} operands,

\[
Y_2 = \{ l_k \in \text{Loc}, 1 \leq k \leq n \mid \text{scope}(l_k) = \alpha.\text{par}(n) \} \cap \text{time}(l_k) = \text{time}(l'_k) + 1 \land \text{scope}(l'_k) = \alpha.\text{par}(n) \}
\]

Let \( Z \) denote the union of the sets \( X_1, X_2, Y_1, Y_2 \), hence

\[ Z = X_1 \cup X_2 \cup Y_1 \cup Y_2 \]

Then, we may capture the rest of the locations along a lifeline in the set,

\[ \text{Loc'} = \text{Loc} \setminus Z \]

We may now give a basic definition that describes how all the rest locations along a lifeline are mapped onto component vectors.

\(|Y|\) is used to denote the cardinality of the set \( Y \).
Definition 3.2.2. Suppose that $\Sigma$ is the signature of a component $c$ represented in a sequence diagram by a lifeline $Cline = (c, Loc, l_0, O_{PE}, SE, RE, Path)$. We define an injective function,

$$vec.map : Loc' \rightarrow \mathcal{P}(V_{SE})$$

given by

- $vec.map(l_0) = \Delta_{SE}$
- $vec.map(l) = \{v_l^{(1)}, v_l^{(2)}, ..., v_l^{(m)}\}$, where $m = |vec.map(l)|$ and $l \in Loc$ such that $time(l) = time(l) - 1$ and for each $j, 1 \leq j \leq m$,

$$v_l^{(j)} = (v_{i_1}^{(j)}, v_{i_2}^{(j)}, ..., v_{i_n}^{(j)})$$

where $n$ is the number of interfaces of $c$ and each coordinate is given by

$$v_{i_j}^{(j)} = \begin{cases} v_{i_j}^{(j)}.e & ; \ (i, e) \in SE \lor (e, l) \in RE \land e \in \beta(l) \\ v_{i_j}^{(j)}, \text{ otherwise} \end{cases}$$

where $1 \leq i \leq n$.

It can be seen that each location $l$ is mapped onto a set of component vectors. The cardinality of the set is that of the set of component vectors associated with its previous location. This might seem somewhat counter-intuitive at this stage, but is necessary because the previous location might have been mapped onto more than one vector. This will be the case when any of the locations preceding $l$ is the end location of a `par` or an `alt` interaction fragment. As will be described in the following section, locations within a `par` potentially execute in parallel while locations within an `alt` are mutually exclusive, thus generating a different execution sequence per operand. The notation used in the definition above is further explained in Figure 3.2.

Definition 3.2.2 works for (and in fact was motivated from) locations appearing in a `seq` interaction fragment. That is to say, it works for all locations along a lifeline that are not in an `alt` or `par` or `sim` fragment. However, it turns out that it can be applied to all locations along a lifeline except for the four cases discussed earlier. It is useful for locations of `sim` and some of `par` and `alt` as will be described shortly. For this reason,
Chapter 3. A Formal Language for Components

Figure 3.2: Notation for component vectors associated location \( l \)

we defer from associating it exclusively with seq (since that would be considerably limiting its applicability).

We demonstrate how sequence diagrams are mapped onto component vectors with a small example.

**Example 3.2.1.** Consider a component \( B \) which interacts with component \( A \) (through interfaces \( i_1 \in P_B \) and \( i_2 \in R_B \)) and component \( D \) (through interface \( i_3 \in R_B \)) in order to perform a certain task. Its signature is given by \( \Sigma_B = (P_B, R_B, \beta_B) \) where \( P_B = \{i_1\}, R_B = \{i_2, i_3\} \) and let \( \beta_B(i_1) = \{a_1, a_2\} \), \( \beta_B(i_2) = \{a_3\} \) and \( \beta_B(i_3) = \{d_1\} \).

We demonstrate how the locations appearing along its lifeline in the sequence diagram of Figure 3.3, are mapped onto component vectors. We write \((x, y, z)\) for \( y \in V_{B_B} \) with \( y(i_1) = x \), \( y(i_2) = y \) and \( y(i_3) = z \). In other words, the first coordinate corresponds to interface \( i_1 \), the second to \( i_2 \) and the third to \( i_3 \).

\( l_0 \) is the initial location and by definition is mapped onto \( \Delta_{\Sigma_B} \). Thus, \( \text{vec.map}(l_0) = (\Delta, \Delta, \Delta) \).

The next location visited is location \( l_1 \), since \( \text{time}(l_1) = \text{time}(l_0) + 1 \). By Definition 3.2.2 we have for this location, \( \text{vec.map}(l_1) = \text{vec.map}(l_0) \). It is mapped onto a single component vector since \( m \) of the definition is \( m = |\text{vec.map}(l_0)| = 1 \) since \( l_0 \) is such that \( \text{time}(l_0) = $
3.2. From UML to the Component Language

Figure 3.3: Example UML 2.0 sequence diagram

\( \text{time}(l_1) - 1 \). The component vector \( v_{l_1} \) is given by

\[
v_{l_1} = (v_{l_{11}}, v_{l_{12}}, v_{l_{13}})
\]

where each coordinate is given by

- \( v_{l_{11}} = v_{l_{01}} \cdot a_1 = a_1 \) since \((a_1, l_1) \in RE_B \land a_1 \in \beta_B(l_1)\)
- \( v_{l_{12}} = v_{l_{02}} = \Lambda \) since \((a_1, l_1) \in RE_B \) but \( a_1 \notin \beta_B(l_2) \)
- \( v_{l_{13}} = v_{l_{03}} = \Lambda \) since \((a_1, l_1) \in RE_B \) but \( a_1 \notin \beta_B(l_3) \)

Hence, \( \text{vec.map}(l_1) = (v_{l_{11}}, v_{l_{12}}, v_{l_{13}}) = (a_1, \Lambda, \Lambda) \).

The next location along the lifeline of component \( B \) is \( l_2 \) (since \( \text{time}(l_2) = \text{time}(l_1) + 1 \)).

By applying Definition 3.2.2, we have for this location \( \text{vec.map}(l_2) = v_{l_2} \), since \( m \) is \( m = |\text{vec.map}(l_1)| = 1 \) and \( l_1 \) is such that \( \text{time}(l_1) = \text{time}(l_2) - 1 \). The component vector \( v_{l_2} \) is given by

\[
v_{l_2} = (v_{l_{21}}, v_{l_{22}}, v_{l_{23}})
\]

where each coordinate is given by

- \( v_{l_{21}} = v_{l_{11}} \cdot a_2 = a_1 a_2 \) since \((a_2, l_2) \in RE_B \land a_2 \in \beta_B(l_1)\)
\begin{itemize}
  \item $v_{l_2} = v_{l_2} = \Lambda$ since $(a_2, l_2) \in RE_B$ but $a_2 \notin \beta_B(i_2)$
  \item $v_{l_3} = v_{l_3} = \Lambda$ since $(a_2, l_2) \in RE_B$ but $a_2 \notin \beta_B(i_3)$
\end{itemize}

Hence, $\text{vec.map}(l_2) = (v_{l_1}, v_{l_2}, v_{l_3}) = (a_1 a_2, \Lambda, \Lambda)$.

The next location along the lifeline of component $B$ is $l_3$ (since $\text{time}(l_3) = \text{time}(l_2) + 1$).

By applying Definition 3.2.2, we have for this location $\text{vec.map}(l_3) = v_{l_3}$, since $m$ of the definition is $m = |\text{vec.map}(l_2)| = 1$ and $l_2$ is such that $\text{time}(l_2) = \text{time}(l_3) - 1$.

The component vector $v_{l_3}$ is given by

\[ v_{l_3} = (v_{l_{l_{13}}}, v_{l_{l_{23}}}, v_{l_{l_{33}}}) \]

where each coordinate is given by

\begin{itemize}
  \item $v_{l_{l_{13}}} = v_{l_{l_{23}}} = a_1 a_2$ since $(l_3, d_1) \in SE_B$ but $d_1 \notin \beta_B(i_1)$
  \item $v_{l_{l_{23}}} = v_{l_{l_{33}}} = \Lambda$ since $(l_3, d_1) \in SE_B$ but $d_1 \notin \beta_B(i_2)$
  \item $v_{l_{l_{33}}} = d_1$ since $(l_3, d_1) \in SE_B \wedge d_1 \in \beta_B(i_3)$
\end{itemize}

Hence, $\text{vec.map}(l_3) = (v_{l_{l_{13}}}, v_{l_{l_{23}}}, v_{l_{l_{33}}}) = (a_1 a_2, \Lambda, d_1)$.

The next location along the lifeline of component $B$ is $l_4$ (since $\text{time}(l_4) = \text{time}(l_3) + 1$).

By applying Definition 3.2.2, we have for this location $\text{vec.map}(l_4) = v_{l_4}$, since $m$ is $m = |\text{vec.map}(l_3)| = 1$ and $l_3$ is such that $\text{time}(l_3) = \text{time}(l_4) - 1$. The component vector $v_{l_4}$ is given by

\[ v_{l_4} = (v_{l_{l_{14}}}, v_{l_{l_{24}}}, v_{l_{l_{34}}}) \]

where each coordinate is given by

\begin{itemize}
  \item $v_{l_{l_{14}}} = v_{l_{l_{24}}} = a_1 a_2$ since $(l_4, a_3) \in SE_B$ but $a_3 \notin \beta_B(i_1)$
  \item $v_{l_{l_{24}}} = v_{l_{l_{34}}} = a_3$ since $(l_4, a_3) \in SE_B \wedge a_3 \in \beta_B(i_2)$
  \item $v_{l_{l_{34}}} = d_1$ since $(l_4, a_3) \in SE_B$ but $a_3 \notin \beta_B(i_3)$
\end{itemize}

Hence, $\text{vec.map}(l_4) = (v_{l_{l_{14}}}, v_{l_{l_{24}}}, v_{l_{l_{34}}}) = (a_1 a_2, a_3, d_1)$. 

\[ \text{vec.map}(l_4) = (v_{l_{l_{14}}}, v_{l_{l_{24}}}, v_{l_{l_{34}}}) = (a_1 a_2, a_3, d_1). \]
3.2.4 Alternative locations

In this section, we extend Definition 3.2.2 to address locations that mark the end of an alt fragment, i.e. \( l \in X_1 \), and the first location of each operand in an alt fragment, i.e. \( l \in Y_1 \). We motivate these extensions as follows.

An alt interaction fragment in a sequence diagram represents choice of behaviour, the choice being between the behaviours described by each of its operands. Recall that at most one of the operands executes [OMG04]. However, the set of execution sequences of the choice is the union of the execution sequences of the operands. Thus, at the end of an alt fragment with \( n \) operands in a diagram there are \( n \) different behaviours, one for each operand. Each of these behaviours or execution sequences arises as a continuation of the start location of the alt fragment.

For this reason, it is important to be able to identify the start location. We have already seen that this can be done by using function \( \text{scope} \) and the corresponding path term, i.e. \( l \) is the start location of an alt interaction fragment with \( n \) operands iff \( \text{scope}(l) = \alpha.\text{alt}(n) \). This use of function \( \text{scope} \) allows us to determine when we come across an alt fragment along a lifeline \( \text{Cline} \) in a sequence diagram.

At the end location of an alt we need to capture the fact that there are \( n \) alternative scenarios the component in question may have engaged in. We do that by associating the end location with \( n \) component vectors corresponding to the last location of each operand. A prerequisite for this idea is that we can determine whether a location is the end location of an alt fragment. In a fashion similar to that for the start location, we make use of function \( \text{scope} \), i.e. \( l \) is the end location of an alt interaction fragment with \( n \) operands iff \( \text{scope}(l) = \overline{\alpha.\text{alt}(n)} \).

For example, once location \( l_1 \) is reached in the sequence diagram of Figure 3.4, component \( B \) has a choice between moving to \( l_2 \) or \( l_4 \). The decision on which one to do is based on evaluation of the condition guarding each operand of the alt fragment. We shall not be concerned with conditions in our approach since they are considered part of the internal processing of the component rather than an event occurring on component interfaces. The outcome of the evaluation of the related condition determines which event will occur next, and this is what is being modelled in our approach. Another
Chapter 3. A Formal Language for Components

possibility is that the choice is resolved by the environment. In our example, the choice is resolved by component A which either does a1 (forcing B to reach location $l_2$) or a2 (forcing B to reach location $l_4$).

Firstly, there is a need to identify the first location of each operand in an alt. This is done by the combined use of functions time and scope as follows. $l$ is the location of the $k$-th operand in an alt interaction fragment with $n$ operands if $\text{time}(l) = \text{time}(ar{l}) + 1$ where $\bar{l}$ is such that $\text{scope}(ar{l}) = \alpha.alt(n)\#(k-1) \vee \text{scope}(ar{l}) = \alpha.alt(n)$. Informally, this says that $l$ is the first location of the $k$-th operand if its previous location $\bar{l}$ (given by function time) belongs to the previous operand (given by function scope) or is the start location of the alt fragment (identified again using function scope).

Secondly, we need to capture the fact that the component vectors associated with the first location of each operand in an alt fragment are obtained based on those of the start location of alt. Put formally,

**Definition 3.2.3.** Let $c$ be a component, with signature $\Sigma = (P, R, \beta)$, represented in a sequence diagram by a lifeline $\text{Cline} = (c, \text{Loc}, l_0, \text{Op}_\Sigma, \text{SE}, \text{RE}, \text{Path})$ and let $l \in \text{Loc}$ be the first location of the $k$-th operand of an alt interaction fragment with $n$ operands on $\text{Cline}$. Then, $\text{vec.map}(l) = \{v_{1}^{(1)}, v_{2}^{(2)}, ..., v_{m}^{(m)}\}$ where $m = |\text{vec.map}(\bar{l})|$ and $\bar{l} \in \text{Loc}$ such that $\text{scope}(\bar{l}) = \alpha.alt(n)$ in which case, for each $j$, $j = 1..m$,

$$v_{i}^{(j)} = \begin{cases} v_{i_{e}}^{(j)} \cdot e, & ((l, e) \in \text{SE} \lor (e, l) \in \text{RE}) \land e \in \beta(i) \\ v_{i}^{(j)}, & \text{otherwise} \end{cases}$$

where $n$ is the number of interfaces of $c$ and each coordinate is given by

This definition applies to locations $l \in Y_1$ with respect to the discussion prior to Definition 3.2.2.

Note that the only difference with respect to Definition 3.2.2 is that in an alt interaction fragment the component vectors associated with the first location of each operand are considered in relation to the start location of the alt fragment instead of its preceding
location. (The first location of the first operand is still considered in relation to its preceding location, but only because this happens to be the start location (of the alt interaction fragment).)

Once the choice has been resolved, execution continues within the chosen operand. When the last location of the operand is reached, execution continues with the end location of the alt fragment. In the sequence diagram of Figure 3.4, if, say, \( l_2 \) is chosen, then \( l_2 \) is visited and then \( l_6 \). If on the other hand, \( l_4 \) is chosen after \( l_1 \), then \( l_5 \) is visited and then \( l_6 \). Hence, at the end location \( l_6 \) we need to capture both alternative scenarios component \( B \) may have participated in before reaching location \( l_6 \). These alternative scenarios are given by the component vectors associated with the last location in each operand, i.e. \( l_3 \) and \( l_5 \) in our example. This is because the component vectors at \( l_3 \) and \( l_5 \) comprise the events that have occurred on component interfaces in each of the alternative scenarios.

Thus, the component vectors associated with an end location of an alt fragment with \( n \) operands are precisely those of the last location of each of the \( n \) operands. This is formally put in the following definition.

**Definition 3.2.4.** Let \( c \) be a component, with signature \( \Sigma = (P, R, \beta) \), represented in a sequence diagram by a lifeline \( CLine = (c, Loc, l_0, Op_1, SE, RE, Path) \) and let \( l \in Loc \) be the end location of an alt interaction fragment with \( n \) operands (i.e. \( \text{scope}(l) = \alpha.\text{alt}(n) \)). Then, the component vectors associated with \( l \) are given by

\[
\text{vec}_{\text{map}}(l) = \bigcup_{k=1}^{n} \text{vec}_{\text{map}}(\hat{l}_k)
\]

where \( \hat{l}_k \), each \( k \), is such that \( \text{scope}(\hat{l}_k) = \alpha.\text{alt}(n)\{k\} \land \text{time}(\hat{l}_k) = \text{time}(\hat{l}) - 1 \) where \( \hat{l} \in Loc \) is such that \( \text{scope}(\hat{l}) = \alpha.\text{alt}(n)\{k+1\} \lor \text{scope}(\hat{l}) = \alpha.\text{alt}(n) \)

This definition applies to location \( l \in X_1 \) with respect to the discussion prior to Definition 3.2.2.

In further explanation of the definition, the last location of each operand is identified by the combination of functions \( \text{time} \) and \( \text{scope} \). Function \( \text{time} \) gives the next location, \( \hat{l}_i \), of \( l_k \), each \( k \), and function \( \text{scope} \) determines whether this next location belongs to the
next operand. The case $\text{scope}(i) = \alpha.\text{alt}(n)$ is included to cater for the last location of the last operand (whose next location is the end of the fragment itself).

Note that there might be some duplication (duplicate component vectors between last location per operand and end location of alt), but there is good reason for it. The component vectors of the last location per operand feed into the resulting component language $V$ (cf Definition 3.2.7) while the component vectors of the end location are used for obtaining the component vectors associated with the location below it (the one visited next).

We demonstrate how the locations appearing within an alt fragment along a component lifeline in a sequence diagram are mapped onto component vectors by means of a small example.

**Example 3.2.2.** We show how the locations appearing along the lifeline of component B in the sequence diagram of Figure 3.4, are mapped onto component vectors.

![Figure 3.4: Example of an alt interaction fragment in a UML 2.0 sequence diagram](image)

The diagram conveys that component B interacts with component A (through interfaces $i_1 \in P_B$ and $i_2 \in R_B$) and component D (through interface $i_3 \in R_B$) in order to
perform a certain task. Assume that its signature is given by $\Sigma_B = (P_B, R_B, \beta_B)$ where $P_B = \{i_1\}$, $R_B = \{i_2, i_3\}$ and $\beta_B(i_1) = \{a_1, a_2\}$, $\beta_B(i_2) = \{a_3\}$ and $\beta_B(i_3) = \{d_1\}$.

Again, we write $(x, y, z)$ for $v \in V_{\Sigma_B}$ with $v(i_1) = x$, $v(i_2) = y$ and $v(i_3) = z$, effectively using the first coordinate for recording events occurring on interface $i_1$, the second for $i_2$ and the third for $i_3$.

$l_0$ is the initial location and by definition is mapped onto $A_{\Sigma_B}$. Thus, $vec\_map(l_0) = (A, A, A)$.

The next location visited is location $l_1$. This location marks the beginning of an alt fragment with 2 operands, since $scope(l_1) = chc.\_alt(2)$. Nevertheless, we still have that $l_1 \in Loc'$ and thus its associated component vectors can be obtained following the construction given in Definition 3.2.2. It is not hard to see that we get $vec\_map(l_1) = (A, A, A)$ which is $vec\_map(l_0)$ since no event is associated with with $l_1$. Hence,

$$vec\_map(l_1) = \psi_1 = (v_{l_1}, v_{l_2}, v_{l_3}) = (A, A, A)$$

The next location considered (not necessarily the one visited next, as explained before) is $l_2$. This is the first location of the 1st operand since $time(l_2) = time(l_1) + 1 \land scope(l_1) = chc.\_alt(2)$. While $scope(l_2) = chc.\_alt(2)|1$. This may be expressed more succinctly by saying that $l_2 \in Y_1$. Thus, its associated component vectors are given by Definition 3.2.3 as follows.

$vec\_map(l_2) = \psi_2$ since $m = |vec\_map(l_1)| = 1$ and $scope(l_1) = chc.\_alt(2)$. The component vector $\psi_2$ is given by

$$\psi_2 = (v_{l_2}, v_{l_2}, v_{l_2})$$

where

- $v_{l_2} = v_{l_1} \cdot a_1 = a_1$ since $(a_1, l_2) \in RE_B \land a_1 \in \beta_B(i_1)$
- $v_{l_2} = v_{l_2} = \Lambda$ since $(a_1, l_2) \in RE_B$ but $a_1 \not\in \beta_B(i_2)$
- $v_{l_2} = v_{l_2} = \Lambda$ since $(a_1, l_2) \in RE_B$ but $a_1 \not\in \beta_B(i_3)$

Hence, $vec\_map(l_2) = (v_{l_2}, v_{l_2}, v_{l_2}) = (a_1, A, A)$. 

The next location visited is $l_3$ (this is compulsory if $l_2$ was visited since $\text{time}(l_3) = \text{time}(l_2) + 1$ and $l_2, l_3$ belong to the same operand). This location belongs to $\text{Loc}'$ and thus its component vectors are given by Definition 3.2.2 as before (in Example 3.2.1). Hence,

$$\text{vec.map}(l_3) = \psi_{l_3} = (v_{i_3}, v_{i_3}, v_{i_3}) = (a_1, A, a_1)$$

The next location considered is $l_4$ (since $\text{time}(l_4) = \text{time}(l_3) + 1$). This is the first location of the 2nd operand of $\text{alt}$, since $\text{time}(l_4) = \text{time}(l_3) + 1 \land \text{scope}(l_3) = \text{chg.alt}(2) \# 1$ while $\text{scope}(l_4) = \text{chg.alt}(2) \# 2$. This may be expressed more succinctly by saying that it belongs to $Y_1$, i.e. $l_4 \in Y_1$. Thus, the component vectors associated with it are given by Definition 3.2.3 as follows.

$$\text{vec.map}(l_4) = \psi_{l_4} \text{ since } m = |\text{vec.map}(l_1)| = 1 \text{ and } \text{scope}(l_1) = \text{chg.alt}(2) \text{. The component vector } \psi_{l_4} \text{ is given by}$$

$$\psi_{l_4} = (v_{i_{l_4}}, v_{i_{l_4}}, v_{i_{l_4}})$$

where

- $v_{i_{l_4}} = v_{i_{l_4}} a_2 = a_2 \text{ since } (a_2, l_4) \in R_E H \land a_2 \notin B(i_1)$
- $v_{i_{l_4}} = v_{i_{l_4}} \Lambda = \Lambda \text{ since } (a_2, l_4) \in R_E B \text{ but } a_2 \notin B(i_2)$
- $v_{i_{l_4}} = v_{i_{l_4}} \Lambda = \Lambda \text{ since } (a_2, l_4) \in R_E B \text{ but } a_2 \notin B(i_3)$

Hence, $\text{vec.map}(l_4) = (v_{i_{l_1}}, v_{i_{l_2}}, v_{i_{l_3}}) = (a_2, A, A)$.

The next location visited is $l_5$ (this is compulsory if $l_4$ was visited since $\text{time}(l_5) = \text{time}(l_4) + 1$ and $l_4, l_5$ belong to the same operand). This location belongs to $\text{Loc}'$ and thus its component vectors are given by Definition 3.2.2 as before (in Example 3.2.1). Hence,

$$\text{vec.map}(l_5) = \psi_{l_5} = (v_{i_5}, v_{i_5}, v_{i_5}) = (a_2, a_3, A)$$

The next location considered is location $l_6$. This is the end location of the $\text{alt}$ fragment, since $\text{scope}(l_6) = \text{chg.alt}(2)$. Thus, the component vectors associated with $l_6$ are given by Definition 3.2.4 as follows.

$$\text{vec.map}(l_6) = \text{vec.map}(l_4) \cup \text{vec.map}(l_5)$$
where \( l_3 \) is the last location of the 1st operand (since \( \text{scope}(l_3) = \text{chc.alt}(2) \| 1 \) \( \text{time}(l_3) = \text{time}(l_4) - 1 \) and \( \text{scope}(l_3) = \text{chc.alt}(2) \| 2 \) and \( l_5 \) is the last location of the second operand (identified using similar reasoning). Hence,

\[
\text{vec.map}(l_6) = \{y^{(1)}_6, y^{(2)}_6\} = \{(a_1, A, d_1), (a_2, a_3, A)\}
\]

The component vector \( y^{(1)}_6 = (a_1, A, d_1) \) describes the behaviour of component \( B \) in case the scenario in the first operand of \( \text{alt} \) is executed while vector \( y^{(2)}_6 = (a_2, a_3, A) \) describes the behaviour resulting from execution of the scenario in the 2nd operand.

It might worth pointing out here that we have not mentioned yet what component vectors go into forming the component language \( V \) of the component in question. We deliberately postpone this until Section 3.2.8.

### 3.2.5 Parallel locations

In this section, we extend Definition 3.2.2 to address locations that mark the end of a \texttt{par} fragment, i.e. \( l \in X_2 \), and the first location of each operand in a \texttt{par} fragment, i.e. \( l \in Y_2 \). We motivate these extensions and give an account of our formal semantics for the \texttt{par} interaction fragment in UML 2.0.

A \texttt{par} interaction fragment in a sequence diagram represents parallelism between the behaviours of its operands. The UML 2.0 specification document [OMG04] hints towards an interleaving semantics, in the sense that the event occurrences of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved. In a certain important sense, this interpretation bears no significance in the UML notation for \texttt{par} and can be understood as a semantic variation point. Note that a formal behavioural semantics is not laid out in the UML specification document. We opt for a non-interleaving or true concurrent semantics for \texttt{par} since it seems a more natural way of thinking about concurrency, on different interfaces of a component. The further abstraction that this semantics provides allows us to also express simultaneity, in the sense of events occurring at exactly the same time.

Simplifying somewhat, instead of considering that locations from different operands are reached in either order, which one would find in an interleaving approach, we consider...
three cases: a) one location is reached first, b) the other location is reached first and
c) both locations are reached at exactly the same time. Another way of expressing
this is by saying that locations from different operands are reached in no particular
order.\footnote{It might be instructive to look at the discussion of Section 4.3.2 with respect to this interpretation.} We return to this different perception of parallelism and give a proper account
of the distinction based on the formal treatment of concurrency within our approach,
in terms of an independence relation in Chapter 4 where we consider the order theoretic
properties of component languages and in terms of pre-order and conflict relations in
Chapter 6 where we consider behavioural presentations for components.

At this stage it suffices to understand that at the end of a \texttt{par} interaction fragment
we must have considered all three cases for each location. For example, once location
\(l_3\) is reached in the sequence diagram of Figure 3.6, we must have considered the case
where \(l_3\) was reached first, the case where \(l_4\) was reached first, and the case where \(l_3\)
and \(l_4\) were reached at exactly the same time.

Recall that events in our model are understood to occur sequentially on a single inter-
face. Concurrent events may only engage distinct interfaces of the component. This
is reflected in our formal treatment of \texttt{par} by imposing the following condition on its
locations. For all \(l_1,l_2 \in \text{Loc}_{\text{par}}\) and \(e_1 \in \beta(i_j), e_2 \in \beta(i_k), 1 \leq j, k \leq n,\) we have

\[
((e_1,l_1) \in RE \lor (l_1,e_1) \in SE)) \land ((e_2,l_2) \in RE \lor (l_2,e_2) \in SE)) \implies j \neq k
\]

Effectively, this says that events (either receive events or send events) associated with
locations in a \texttt{par} interaction fragment belong to different interfaces of the component.

Since the event occurrences of different operands are independent of each other, the
resulting behaviours of each operand arise as continuations of the start location of the
\texttt{par} fragment (in a fashion similar to the \texttt{alt} fragment). Consequently, it is important
to be able to identify the start location of a \texttt{par} fragment. This is done in our approach
using function \texttt{scope} and the appropriate path term, i.e. \(l\) is the start location of a \texttt{par}
fragment with \(n\) operands iff \(\text{scope}(l) = \alpha.par(n)\).

There is also a need to identify the first location of each operand. Similarly to the
\texttt{alt} fragment, this is done by use of functions \texttt{time} and \texttt{scope} as follows. \(l\) is the first
location of the k-th operand in a par interaction fragment with n operands iff \( \text{time}(l) = \text{time}(\bar{l}) + 1 \) where \( \bar{l} \) is such that \( \text{scope}(\bar{l}) = \alpha.\text{par}(n) \cup ((k - 1) \lor \text{scope}(\bar{l}) = \alpha.\text{par}(n)) \).

Additionally, we need to capture the fact that the component vectors associated with the first location of each operand in a par fragment are obtained based on those of the start location of the fragment. Put formally,

**Definition 3.2.5.** Let \( c \) be a component, with signature \( \Sigma = (P,R,\beta) \), represented in a sequence diagram by a lifeline \( \text{Cline} = (c, \text{Loc}, l_0, \text{Op} \subseteq \text{SE}, \text{RE}, \text{Path}) \) and let \( l \in \text{Loc} \) be the first location of the k-th operand of a par interaction fragment with n operands on Cline. Then, \( \text{vec.map}(l) = \{ \underline{v}_{1}^{(1)}, \underline{v}_{2}^{(2)}, \ldots, \underline{v}_{n}^{(m)} \} \) where \( m = |\text{vec.map}(l)| \) and \( l \in \text{Loc} \) such that \( \text{scope}(\bar{l}) = \alpha.\text{par}(n) \) in which case, for each \( j, j = 1..m \),

\[
\underline{v}_{j}^{(j)} = (v_{1}^{(j)}, v_{2}^{(j)}, \ldots, v_{n}^{(j)})
\]

where \( n \) is the number of interfaces of \( c \) and each coordinate is given by

\[
v_{i}^{(j)} = \begin{cases} v_{i}^{(j)}, e, & (l, e) \in \text{SE} \lor (e, l) \in \text{RE} \land e \in \beta(i) \\
v_{i}^{(j)}, & \text{otherwise}
\end{cases}
\]

where \( i = 1..n \).

This definition applies to locations \( l \in \text{Y}_{2} \) with respect to the discussion prior to Definition 3.2.2. It is essentially an adaptation of Definition 3.2.3 for par.

All locations that appear in a par fragment, other than the first location of each operand or the end location of the fragment, belong to the set \( \text{Loc'} \) and their associated component vectors are given by Definition 3.2.2.

We now turn our attention to the end location of a par interaction fragment. The component vectors associated with the end location \( l \) of a par are of particular importance since, as we will see, \( \text{vec.map}(l) \) is what actually feeds into the corresponding component language \( V \) from the whole par fragment (cf Definition 3.2.7).

In what follows, we describe the three cases that need to be considered for each location appearing within a par fragment so as to reflect the fact that the event occurrences appearing in different operands are effectively unordered (in parallel). For each location
in \texttt{par} excluding those of the last operand (since they will have already been considered in relation to all locations of the preceding operands), we need to compute the three cases mentioned above. We formulate each of them below.

For each location $l \in \text{Loc}_\text{par}$ such that $\text{scope}(l) \neq \alpha.\text{par}(n)$ determine,

- $\text{vec.map}(l)^I = \text{vec.map}(l)$.
  This gives the component vectors associated with $l$, when $l$ is reached first.

- $\text{vec.map}(l)^{II} = \bigcup_{s=1}^{X} \text{vec.map}(\bar{l}_s)$ where
  
  $X = \{\bar{l} \in \text{Loc}_\text{par} \mid \text{scope}(\bar{l}) = \alpha.\text{par}(n) \& j, (b+1) \leq j \leq n\}$

  This gives the component vectors associated with $l$ when all other locations $\bar{l}$ of the succeeding operands are reached first (before reaching $l$).

- $\text{vec.map}(l)^{III} = \{\underline{u}_l^{(j)} = \underline{u}_l^{(j)} \cup \underline{u}_l^{(j)} \mid \forall \bar{l} : \text{scope}(\bar{l}) = \alpha.\text{par}(n) \& j, (k+1) \leq j \leq n\}$
  where $1 \leq j \leq m$ and $m = |\text{vec.map}(l)|$ where $\bar{l}$ is such that $\text{scope}(\bar{l}) = \alpha.\text{par}(n)$.

  This gives the component vectors associated with location $l$ of the $k$-th operand when it is reached at exactly the same time with another location $\bar{l}$ from a different (and succeeding) operand. The superscript $j$ runs through the $m$ component vectors associated with location $l$, where $m$ is as before (Definition 3.2.2).

The operation $\cup$ on component vectors gives their least upper bound and is formally defined in Chapter 4 (cf Definition 4.1.3) where we study the order theoretic properties of component vectors. For the purpose of computing $\text{vec.map}(l)^{III}$ it suffices to understand that, given two component vectors $\underline{u}_1$ and $\underline{u}_2$, the resulting component vector $\underline{u}_1 \cup \underline{u}_2$ is obtained by comparing their coordinates pairwise and keeping the one whose sequence is of greater length. This is demonstrated in Example 3.2.3 below.

By considering the above three cases, each location $l$ in a \texttt{par} fragment is mapped onto three (sets of) component vectors, $\text{vec.map}(l)^I$, $\text{vec.map}(l)^{II}$ and $\text{vec.map}(l)^{III}$. Now the end location of \texttt{par} is associated with the component vectors (including all three cases) corresponding to each location appearing within the fragment. This is formally put in the following definition.
Definition 3.2.6. Let $c$ be a component, with signature $\Sigma = (P, R, \beta)$, represented in a sequence diagram by a lifeline Cline $= (c, Loc, i_0, Op_2, SE, RE, Path)$ and let $l \in Loc$ be the end location of a par interaction fragment with $n$ operands. Then, the component vectors associated with $l$ are given by,

$$vec\_map(l) = \bigcup_{r=1}^{|Y|} vec\_map(l_r)^x$$

where $Y = \{l \in Loc_{par} | scope(l) = \alpha.par(n)\|k, 1 \leq k \leq (n - 1)\}$ and $x = I, II, III$.

Note that we do not consider locations of the last operand in the definition because they will have already been considered in going through the preceding $n - 1$ operands. This definition is an extension of Definition 3.2.2 that applies to locations $l \in X_2$.

In further explanation of the notation we have included Figure 3.5.

![Figure 3.5: Notation for component vectors of the end location of par](image)

Example 3.2.3. We demonstrate how the locations appearing along the lifeline of component $B$ in the sequence diagram of Figure 3.6 are mapped onto component vectors.

Consider a component $B$ which, in response to an event $a1$ from component $A$, requires services from components $D$ and $E$ concurrently by generating events $d1$ and $e1$ on the respective interfaces, as shown in the interaction described by the sequence diagram in Figure 3.6. Let $\Sigma_B$ be the signature of component $B$ given by $\Sigma_B = (P_B, R_B, \beta_B)$ where $P_B = \{i_1\}$, $R_B = \{i_2, i_3\}$ and $\beta_B(i_1) = \{a1\}$, $\beta_B(i_2) = \{d1\}$ and $\beta_B(i_3) = \{e1\}$. We write $(x, y, z)$ for component vectors in $V_{\Sigma_B}$ where $x = y(i_1)$, $y = y(i_2)$ and $z = y(i_3)$. In other words, the first coordinate of the component vectors corresponds to the interface $i_1$, the second to $i_2$ and the third to $i_3$. 

---

*Note: The image referenced in the text is not provided in the current text.*
Figure 3.6: Example of a `par` interaction fragment in a UML 2.0 sequence diagram

$l_0$ is the initial location and by Definition 3.2.2 is mapped onto $\Delta_{\Sigma_B}$. Thus, vec.map($l_0$) = $(\Lambda, \Lambda, \Lambda)$.

The next location reached is location $l_1$ since $\text{time}(l_1) = \text{time}(l_0) + 1$. This location belongs to $\text{Loc}^c$ and thus by Definition 3.2.2 we have,

$$\text{vec.map}(l_1) = v_{l_1} = (v_{l_{11}}, v_{l_{12}}, v_{l_{13}}) = (a1, \Lambda, \Lambda)$$

The next location reached is location $l_2$ (since $\text{time}(l_2) = \text{time}(l_1) + 1$). This location marks the beginning of the `par` fragment with 2 operands, since $\text{scope}(l_2) = \text{prl.par}(2)$. Nevertheless, it belongs to $\text{Loc}^c$ and thus by applying Definition 3.2.2, we have

$$\text{vec.map}(l_2) = v_{l_2} = (v_{l_21}, v_{l_22}, v_{l_23}) = (a1, \Lambda, \Lambda)$$

The next location considered (not necessarily the next one to be reached, as explained before) is $l_3$. This is the first location of the 1st operand since $\text{time}(l_3) = \text{time}(l_2) + 1 \land \text{scope}(l_2) = \text{prl.par}(2)$ while $\text{scope}(l_3) = \text{prl.par}(2) \uparrow 1$. This may be expressed more succinctly by saying that $l_3 \in Y_2$. Thus, its associated component vectors are given by Definition 3.2.5 as follows.
3.2. From UML to the Component Language

vec_map(l_3) = v_{l_3} since m = |vec_map(l_2)| = 1 and scope(l_3) = pri.par(2). The component vector $v_{l_3}$ is given by

$$v_{l_3} = (v_{l_3}, v_{l_3}, v_{l_3})$$

where

- $v_{l_3} = v_{l_3} = a_1$ since $(l_3, d_1) \in SE_B$ but $d_1 \not\in B(i_1)$
- $v_{l_3} = v_{l_3}.d_1 = d_1$ since $(l_3, d_1) \in SE_B \land d_1 \in B(i_2)$
- $v_{l_3} = v_{l_3} = \Lambda$ since $(l_3, d_1) \in SE_B$ but $d_1 \not\in B(i_3)$

Hence, vec_map(l_3) = (v_{l_3}, v_{l_3}, v_{l_3}) = (a_1, d_1, \Lambda).

The next location considered (not necessarily the next one to be reached, as explained before) is $l_4$. This is the first location of the 2nd operand, since time(l_4) = time(l_3) + 1 \land scope(l_3) = pri.par(2)[1] while scope(l_4) = pri.par(2)[2]. This may be expressed more succinctly by saying that $l_4 \in Y_2$. Thus, its associated component vectors are given by Definition 3.2.5 as follows.

vec_map(l_4) = v_{l_4} since m = |vec_map(l_2)| = 1 and scope(l_3) = pri.par(2). The component vector $v_{l_4}$ is given by

$$v_{l_4} = (v_{l_4}, v_{l_4}, v_{l_4})$$

where

- $v_{l_4} = v_{l_4} = a_1$ since $(l_4, e_1) \in SE_B$ but $e_1 \not\in B(i_1)$
- $v_{l_4} = v_{l_4} = \Lambda$ since $(l_4, e_1) \in SE_B$ but $e_1 \not\in B(i_2)$
- $v_{l_4} = v_{l_4}.e_1 = e_1$ since $(l_4, e_1) \in SE_B \land e_1 \in B(i_3)$

Hence, vec_map(l_4) = (v_{l_4}, v_{l_4}, v_{l_4}) = (a_1, \Lambda, e_1).

The next location considered is location $l_5$. This is the end location of the par fragment, i.e. $l_5 \in X_2$, since scope(l_5) = pri.par(2). Thus, the component vectors associated with it are given by Definition 3.2.6 as follows.
First, we determine the three cases for each location appearing within the two operands of \texttt{par}, i.e. all \( l \) such that \( l \in \text{Loc}_{\text{par}} \). In fact, we have seen that we do not need to consider locations of the last operand of \texttt{par}. So, in our example it suffices to determine the three cases for location \( l_3 \) only, that is, \( \text{vec.map}(l_3)^I \), \( \text{vec.map}(l_3)^{II} \) and \( \text{vec.map}(l_3)^{III} \).

- \( \text{vec.map}(l_3)^I = \text{vec.map}(l_3) = (a_1, d_1, \Lambda) \)

- \( \text{vec.map}(l_3)^{II} = \text{vec.map}(l_4) = (a_1, \Lambda, e_1) \)

since \( l_4 \) is the only location appearing in the succeeding operands, which in our example is only one, the 2nd operand of the \texttt{par} fragment.

- \( \text{vec.map}(l_3)^{III} = u_{l_5,l_4} \)

since \( m = |\text{vec.map}(l_2)| = 1 \) where \( l_2 \) is the start location of \texttt{par}, and \( l_4 \) is the only location such that \( \text{scope}(l_4) = \text{prl.par}(2)\parallel 2 \). The corresponding component vector \( u_{l_5,l_4} \) is given by

\[
u_{l_5,l_4} = u_{l_5} \cup u_{l_4} = (a_1, d_1, \Lambda) \cup (a_1, \Lambda, e_1) = (a_1, d_1, e_1)
\]

Hence, \( \text{vec.map}(l_3)^{III} = (a_1, d_1, e_1) \)

We may now obtain the component vectors associated with the end location \( l_5 \) of \texttt{par} by applying Definition 3.2.6.

\[
\text{vec.map}(l_5) = \text{vec.map}(l_3)^I \cup \text{vec.map}(l_3)^{II} \cup \text{vec.map}(l_3)^{III} = \{(a_1, d_1, \Lambda), (a_1, \Lambda, e_1), (a_1, d_1, e_1)\}
\]

since \( l_3 \) is the only location appearing in an operand which is not the last operand of the \texttt{par} fragment.

Thus, the component vectors associated with the locations of the \texttt{par} fragment in the sequence diagram of Figure 3.6 are,

\[
(a_1, \Lambda, \Lambda), (a_1, d_1, \Lambda), (a_1, \Lambda, e_1), (a_1, d_1, e_1)
\]
Notice that in terms of event occurrences within the par fragment the second vector captures the case that event d1 occurs first, the third vector the case where event c1 occurs first, and the fourth vector the case that events c1 and d1 occur at exactly the same time. This reflects the fact that locations \( l_4, l_5 \) are unordered in the sequence diagram of Figure 3.6.

A more interesting par interaction fragment is addressed in the case study described in Section 3.3. A par fragment which contains an alt fragment within one of its operands is addressed in Appendix A.

3.2.6 Simultaneous locations

In this section, we describe how we go about mapping locations appearing in a sim interaction fragment onto component vectors. A sim fragment represents the simultaneous occurrence of all events appearing in it. Recall that sim has a unique operand, the fragment itself. The intuition is that events in a sim have to happen at the same time.

Further, events in our model can only occur sequentially on a single interface. Concurrent or simultaneous events necessarily engage distinct interfaces of the component. In similar fashion to par, this is reflected in our formal treatment of sim by imposing the following condition on its locations. For all \( l_1, l_2 \in \text{Loc}_\text{sim} \) and \( e_1 \in \beta(i_j), e_2 \in \beta(i_k), 1 \leq j, k \leq n, \) we have

\[
((e_1, l_1) \in RE \lor (l_1, e_1) \in SE) \land ((e_2, l_2) \in RE \lor (l_2, e_2) \in SE) \implies j \neq k
\]

Effectively, this says that events (either receive events or send events) associated with locations in a sim interaction fragment belong to different interfaces of the component.

It turns out that simultaneous locations can be handled with what has been presented so far about the function vec.map, so no further extensions to Definition 3.2.2 are required. We justify this position and explain how simultaneity is captured, in the remaining of this section.

The start location of sim is identified by function scope (as before). That is, location \( l \) is the start location of a sim interaction fragment iff \( \text{scope}(l) = \alpha . \text{sim} \).
The behaviours, captured by component vectors in our approach, corresponding to locations within a \texttt{sim} fragment are treated atomically (all at once) and cannot be interleaved by behaviours corresponding to other locations. However, the component vectors associated with each location in \texttt{sim} can be obtained by applying Definition 3.2.2, as if they were in the main fragment of the sequence diagram.

The fact that the \texttt{sim} fragment is treated atomically by the enclosing fragment (or the sequence diagram as a whole) is captured as follows. Each location within a \texttt{sim} is mapped onto (a set of) component vectors following the construction given in Definition 3.2.2. This implies that the end location of the fragment will be mapped onto (a set of) component vectors in which all event occurrences associated with locations within \texttt{sim} appear. The fact that these event occurrences are simultaneous is captured by ignoring all component vectors associated with these locations and keeping only the vectors associated with the end location of the \texttt{sim} fragment.

We illustrate the idea by means of a small example.

**Example 3.2.4.** Consider a component $B$ which, in response to receiving event $a1$ from component $A$, requires services from components $D$ and $E$ simultaneously by generating events $d1$ and $e1$ on their respective interfaces at the same time. The interaction is described in the sequence diagram of Figure 3.7.

Let $\Sigma_B$ be the signature of component $B$ given by $\Sigma_B = (P_B, R_B, \beta_B)$ where $P_B = \{i_1\}$, $R_B = \{i_2, i_3\}$ and $\beta_B(i_1) = \{a1\}$, $\beta_B(i_2) = \{d1\}$ and $\beta_B(i_3) = \{e1\}$. We write $(x, y, z)$ for component vectors in $V_{\Sigma_B}$ where $x = \nu(i_1)$, $y = \nu(i_2)$ and $z = \nu(i_3)$. In other words, the first coordinate of the component vectors corresponds to the interface $i_1$, the second to $i_2$ and the third to $i_3$.

\(l_0\) is the initial location and by Definition 3.2.2 is mapped onto $\Lambda_{\Sigma_B}$. Thus, $\text{vec.map}(l_0) = (\Lambda, \Lambda, \Lambda)$.

The next location reached is location $l_1$ since $\text{time}(l_1) = \text{time}(l_0) + 1$. This location belongs to $\text{Loc'}$ and thus by Definition 3.2.2 we have, $\text{vec.map}(l_1) = (a1, \Lambda, \Lambda)$.

The next location reached is location $l_2$ (since $\text{time}(l_2) = \text{time}(l_1) + 1$). This location marks the beginning of the \texttt{sim} fragment, since $\text{scope}(l_2) = \text{smt.sim}$. Nevertheless, it
Figure 3.7: Example of a sim interaction fragment in a UML 2.0 sequence diagram

belongs to Loc' and thus by applying Definition 3.2.2, we have

\[ \text{vec.map}(l_2) = \psi_{l_2} = (v_{i_{21}}, v_{i_{22}}, v_{i_{23}}) = (a_1, \Lambda, \Lambda) \]

The next location considered (but not the next one to be reached, at least not on its own) is \( l_3 \), since \( \text{time}(l_3) = \text{time}(l_2) + 1 \). This location belongs to \( \text{Loc}' \) and its corresponding component vectors are given by Definition 3.2.2. It is not hard to see that this gives

\[ \text{vec.map}(l_3) = \psi_{l_3} = (v_{i_{31}}, v_{i_{32}}, v_{i_{33}}) = (a_1, d_1, \Lambda) \]

The next location considered is \( l_4 \) (since \( \text{time}(l_4) = \text{time}(l_3) + 1 \)). This location also belongs to \( \text{Loc}' \) and is treated as such by Definition 3.2.2. Thus, we have

\[ \text{vec.map}(l_4) = \psi_{l_4} \text{ since } m = |\text{vec.map}(l_3)| = 1 \text{ where } \text{time}(l_3) = \text{time}(l_4) - 1. \]

The component vector \( \psi_{l_4} \) is given by

\[ \psi_{l_4} = (v_{i_{41}}, v_{i_{42}}, v_{i_{43}}) \]

where

- \( v_{i_{41}} = v_{i_{31}} = a_1 \text{ since } (l_4, e_1) \in SE_B \text{ but } e_1 \not\in \beta_B(i_1) \)
- \( v_{i_{42}} = v_{i_{32}} = d_1 \text{ since } (l_4, e_1) \in SE_B \text{ but } e_1 \not\in \beta_B(i_2) \)
• \(v_{l_5} = v_{l_5} \cdot e_1 = e_1\) since \((l_4, e_1) \in SE_B \land e_1 \in \beta_B(l_5)\)

Hence, \(\text{vec.map}(l_4) = (v_{l_4}, v_{l_5}, v_{l_5}) = (a_1, d_1, e_1)\).

The next location considered is \(l_5\). This location marks the end of the sim fragment since \(\text{scope}(l_5) = \text{siml.sim}\). Nevertheless, it still belongs to \(\text{Loc'}\) and thus its associated component vectors are also given by Definition 3.2.2 as follows.

\[
\text{vec.map}(l_5) = \psi_{l_5} \quad \text{since} \quad m = |\text{vec.map}(l_4)| = 1 \quad \text{where} \quad \text{time}(l_4) = \text{time}(l_5) - 1.
\]

The component vector \(\psi_{l_5}\) is given by,

\[
\psi_{l_5} = (v_{l_5}, v_{l_5}, v_{l_5})
\]

where

• \(v_{l_5} = v_{l_5} = a_1\) since \(\exists e \in Op_{SE_B}\) such that \((l_5, e) \in SE_B \lor (e, l_5) \in RE_B\)

• \(v_{l_5} = v_{l_5} = d_1\) since \(\exists e \in Op_{SE_B}\) such that \((l_5, e) \in SE_B \lor (e, l_5) \in RE_B\)

• \(v_{l_5} = v_{l_5} = e_1\) since \(\exists e \in Op_{SE_B}\) such that \((l_5, e) \in SE_B \lor (e, l_5) \in RE_B\)

Hence, \(\text{vec.map}(l_5) = (v_{l_5}, v_{l_5}, v_{l_5}) = (a_1, d_1, e_1)\).

Note that this is the same as \(\text{vec.map}(l_4)\) but this was expected since no event is associated with location \(l_5\).

As prescribed earlier, what we keep from the whole sim fragment is the component vectors associated with its end location only. That is, \(\text{vec.map}(l_5)\) in our example.

Effectively, we are throwing away the component vectors associated with locations appearing within the sim fragment, i.e. \(\text{vec.map}(l_3)\) and \(\text{vec.map}(l_4)\). We shall see how this is enforced in Section 3.2.8 where we define the set of component vectors associated with a component lifeline as a whole.

In this way, the behaviours associated with the sim fragment in Figure 3.7 are,

\[(a_1, \Lambda, \Lambda), (a_1, d_1, e_1)\]

The first component vector corresponds to the start location of sim and the second to its end location. It can be seen that what takes us from one to the other is the simultaneous occurrence of events \(e_1\) and \(d_1\).
The interested reader might care to compare the component vectors resulting from considering the \texttt{par} fragment in Example 3.2.3 and the component vectors resulting from considering the \texttt{sim} fragment in the example given above. The comparison provides useful insights on how the relationships between events manifest themselves in the order structure of the corresponding component language. This is discussed in detail in the next chapter (Section 4.3).

3.2.7 Nested locations

It is often the case in a sequence diagram that interaction fragments are used within some operand of another interaction fragment in order to capture more complex aspects of behaviour. We refer to locations appearing in such parts of a sequence diagram as \textit{nested} locations.

In our approach nested locations are treated \textit{inside-out}. That is, locations are first mapped onto component vectors based on the interaction fragment that contains them, and then based on the enclosing fragment. By enclosing fragment we refer to the interaction fragment that immediately contains another interaction fragment in one of its operands. By simply addressing nested locations \textit{inside-out}, there is no need for further extensions to the formal construction described so far.

The idea may be best illustrated through an example. In the following we show how locations of an \texttt{alt} fragment (nested fragment) appearing within an operand of another \texttt{alt} fragment (enclosing fragment) are mapped onto component vectors.

\textbf{Example 3.2.5.} Consider a component $B$ which participates in the interaction described in the UML 2.0 sequence diagram of Figure 3.8 in order to achieve a certain task. Let $\Sigma_B$ be the signature of component $B$ given by $\Sigma_B = (P_B, R_B, \beta_B)$, where $P_B = \{i_1\}$, $R_B = \{i_2, i_3\}$ and $\beta_B(i_1) = \{a_1, a_2, a_3\}$, $\beta_B(i_2) = \{a_4\}$ and $\beta_B(i_3) = \{d_1, d_2\}$.

We write $(x, y, z)$ for component vectors in $V_{\Sigma_B}$ where $x = \nu(i_1)$, $y = \nu(i_2)$ and $z = \nu(i_3)$. In other words, the first coordinate of the component vectors corresponds to the interface $i_1$, the second to $i_2$ and the third to $i_3$.

$l_0$ is the initial location and by Definition 3.2.2 is mapped onto $A_{\Sigma_B}$. Thus, \texttt{vec-map}(l_0) = (A, A, A).
Figure 3.8: Example of an alt fragment within another alt fragment

The next location reached is location \( l_1 \) since \( \text{time}(l_1) = \text{time}(l_0) + 1 \). This location belongs to \( \text{Loc}' \) and thus by Definition 3.2.2 we have,

\[
\text{vec.map}(l_1) = 2_{l_1} = (v_{l_1}, v_{l_2}, v_{l_3}) = (a1, A, A)
\]

The next location reached is location \( l_2 \). This location marks the beginning of an \textit{alt} fragment with 2 operands, since \( \text{scope}(l_2) = \text{nchc.alt}(2) \). Nevertheless, we still have that \( l_2 \in \text{Loc}' \) and thus its associated component vectors can be obtained following the construction given in Definition 3.2.2. It is not hard to see that we get \( \text{vec.map}(l_2) = \)
3.2. From UML to the Component Language

\((a_1, A, A)\) which is \(\text{vec.map}(l_1)\) since no event is associated with \(l_2\). Hence,

\[\text{vec.map}(l_2) = u_{i_2} = (v_{i_2}, v_{i_2}, v_{i_2}) = (a_1, A, A)\]

The next location considered (not necessarily the next one to be reached, since \(l_5\) may also be reached directly from \(l_2\) as explained before) is \(l_3\). This is the first location of the 1st operand since \(\text{time}(l_3) = \text{time}(l_2) + 1 \land \text{scope}(l_3) = \text{nchc.alt}(2) \| 1\) while \(\text{scope}(l_2) = \text{nchc.alt}(2)\). This may be expressed more succinctly by saying that \(l_3 \in Y_1\). Thus, its associated component vectors are given by Definition 3.2.3 as follows.

\[\text{vec.map}(l_3) = u_{i_3}\] since \(m = |\text{vec.map}(l_2)| = 1\) and \(\text{scope}(l_2) = \text{nchc.alt}(2)\). The component vector \(u_{i_3}\) is given by

\[u_{i_3} = (v_{i_3}, v_{i_3}, v_{i_3})\]

where

- \(v_{i_3} = v_{i_2} - a_2 = a_1a_2\) since \((a_2, l_3) \in \text{RE}_B \land a_2 \in \beta_B(i_1)\)
- \(v_{i_3} = v_{i_2} - A\) since \((a_2, l_3) \in \text{RE}_B\) but \(a_2 \notin \beta_B(i_2)\)
- \(v_{i_3} = v_{i_2} - A\) since \((a_2, l_3) \in \text{RE}_B\) but \(a_2 \notin \beta_B(i_3)\)

Hence, \(\text{vec.map}(l_3) = (u_{i_3}, v_{i_3}, v_{i_3}) = (a_1a_2, A, A)\).

The next location reached (and this is compulsory if \(l_3\) was visited, since \(\text{time}(l_4) = \text{time}(l_3) + 1\) and \(l_3, l_4\) belong to the same operand) is \(l_4\). This location belongs to Loc' and thus its component vectors are given by Definition 3.2.2 as before (in Example 3.2.1). Hence,

\[\text{vec.map}(l_4) = u_{i_4} = (v_{i_4}, v_{i_4}, v_{i_4}) = (a_1a_2, A, d1)\]

The next location considered (not necessarily the next one to be reached, as explained before) is \(l_5\). This is the first location of the 2nd operand, since \(\text{time}(l_5) = \text{time}(l_4) + 1 \land \text{scope}(l_4) = \text{nchc.alt}(2) \| 1\) while \(\text{scope}(l_5) = \text{nchc.alt}(2) \| 2\). This may be expressed more succinctly by saying that \(l_5 \in Y_1\). Thus, its associated component vectors are given by Definition 3.2.3 as follows.
vec.map(l₃) = v₅ since m = |vec.map(l₃)| = 1 and scope(l₃) = nch.c.alt(2). The component vector v₅ is given by

\[ \begin{align*}
\mathbb{v}_5 &= (v_{i_{01}}, v_{i_{02}}, v_{i_{03}}) \\
\end{align*} \]

where

- \[ v_{i_{01}} = v_{i_{02}} = a3 = 1 \text{ since } (a3, l₃) \in RE_B \land a3 \in \beta_B(i₁) \]
- \[ v_{i_{02}} = v_{i_{03}} = \Lambda \text{ since } (a3, l₃) \in RE_B \text{ but } a3 \notin \beta_B(i₂) \]
- \[ v_{i_{03}} = v_{i_{03}} = \Lambda \text{ since } (a3, l₃) \in RE_B \text{ but } a3 \notin \beta_B(i₃) \]

Hence, vec.map(l₃) = (v_{i_{01}}, v_{i_{02}}, v_{i_{03}}) = (a1a3, \Lambda, \Lambda).

The next location reached (and this is compulsory since l₅ was visited and time(l₆) = time(l₅) + 1 and l₅, l₆ belong to the same operand) is l₆. This location marks the beginning of a new alt fragment with 2 operands, since scope(l₆) = nch.c.alt(2) \| 2.alt(2). Nevertheless, we still have that l₆ \notin Loc' and thus its associated component vectors can be obtained following the construction given in Definition 3.2.2. It is not hard to see that we get

vec.map(l₆) = v₆ = (v_{i₆₁}, v_{i₆₂}, v_{i₆₃}) = (a1a3, \Lambda, \Lambda)

which is vec.map(l₆) since no event is associated with with l₆. Hence, vec.map(l₆) = (a1a3, \Lambda, \Lambda).

Note that l₆ is the start location for the nested alt fragment. This is important in determining the component vectors for the first location of each operand of the nested alt fragment.

The next location considered (not necessarily the next one to be reached, as explained before) is l₇. This is the first location of the 1st operand of the nested alt since time(l₇) = time(l₆) + 1 \land scope(l₇) = nch.alt(2) \| 2.alt(2) while scope(l₇) = nch.alt(2) \| 2.alt(2) \| 1. This may be expressed more succinctly by saying that l₇ \notin Y₁. Thus, its associated component vectors are given by Definition 3.2.3 as follows.

vec.map(l₇) = v₇ since m = |vec.map(l₆)| = 1 and scope(l₇) = nch.alt(2) \| 1.alt(2). The component vector v₇ is given by

\[ \begin{align*}
\mathbb{v}_7 &= (v_{i₇₁}, v_{i₇₂}, v_{i₇₃}) \\
\end{align*} \]
3.2. From UML to the Component Language

where

- \( v_{tr1} = v_{l_{10}} = a1a3 \) since \((l_7, a4) \in SE_B \) but \( a4 \notin \beta_B(i_1) \)
- \( v_{tr2} = v_{l_{12}} = a4 \) since \((l_7, a4) \in SE_B \land a4 \in \beta_B(i_2) \)
- \( v_{tr3} = v_{l_{13}} = \Lambda \) since \((l_7, a4) \in SE_B \) but \( a4 \notin \beta_B(i_3) \)

Hence, \( \text{vec.map}(l_7) = (v_{l_{11}}, v_{l_{12}}, v_{l_{13}}) = (a1a3, a4, \Lambda) \).

The next location considered is \( l_8 \). This is the first location of the 2nd operand of the nested alt since \( \text{time}(l_6) = \text{time}(l_7) + 1 \land \text{scope}(l_7) = \text{nchc.alt}(2)\|2.alt(2)\|1 \) while \( \text{scope}(l_8) = \text{nchc.alt}(2)\|2.alt(2)\|2 \). This may be expressed more succinctly by saying that \( l_8 \in Y_7 \). Thus, its associated component vectors are given by Definition 3.2.3 as follows.

\[ \text{vec.map}(l_8) = v_{l_8} \] since \( m = \left| \text{vec.map}(l_6) \right| = 1 \) and \( \text{scope}(l_6) = \text{nchc.alt}(2)\|2.alt(2) \).

The component vector \( v_{l_8} \) is given by

\[ v_{l_8} = (v_{l_{11}}, v_{l_{12}}, v_{l_{13}}) \]

where

- \( v_{l_{11}} = v_{l_{12}} = a1a3 \) since \((l_8, d2) \in SE_B \) but \( d2 \notin \beta_B(i_1) \)
- \( v_{l_{12}} = v_{l_{13}} = \Lambda \) since \((l_8, d2) \in SE_B \) but \( d2 \notin \beta_B(i_2) \)
- \( v_{l_{13}} = v_{l_{13}} = d2 \) since \((l_8, d2) \in SE_B \land d2 \in \beta_B(i_3) \)

Hence, \( \text{vec.map}(l_8) = (v_{l_{11}}, v_{l_{12}}, v_{l_{13}}) = (a1a3, \Lambda, d2) \).

Note that the component vectors of \( l_7, l_8 \) were computed based on those of \( l_6 \) since \( l_6 \) is the start location of the nested alt.

The next location considered is location \( l_9 \). This is the end location of the nested alt fragment, since \( \text{scope}(l_9) = \text{nchc.alt}(2)\|2.alt(2) \). Thus, the component vectors associated with \( l_9 \) are given by Definition 3.2.4 as follows:

\[ \text{vec.map}(l_9) = \text{vec.map}(l_7) \cup \text{vec.map}(l_8) \]
where $l_7$ is the last location of the 1st operand (since $\text{time}(l_7) = \text{time}(l_8) - 1 \land \text{scope}(l_8) = \text{nchc}.alt(2)\|2.alt(2)\|2$ while $\text{scope}(l_7) = \text{nchc}.alt(2)\|2.alt(2)\|1$) and $l_8$ is the last location of the 2nd operand (identified using similar reasoning). Hence,

$$\text{vec.map}(l_9) = \{v_{l_9}^{(1)}, v_{l_9}^{(2)}\} = \{(a1a3, a4, \Lambda), (a1a3, \Lambda, d2)\}$$

The component vector $(a1a3, a4, \Lambda)$ describes the behaviour of component $B$ in case the scenario in the first operand of the nested alt is executed while vector $(a1a3, \Lambda, d2)$ describes the behaviour resulting from execution of its 2nd operand.

The next location considered is location $l_{10}$. This is the end location of the enclosing alt fragment, since $\text{scope}(l_{10}) = \text{nchc}.alt(2)$. Thus, the component vectors associated with $l_{10}$ are given by Definition 3.2.4 as follows.

$$\text{vec.map}(l_{10}) = \text{vec.map}(l_4) \cup \text{vec.map}(l_9)$$

where $l_4$ is the last location of the 1st operand (since $\text{scope}(l_4) = \text{nchc}.alt(2)\|1 \land \text{time}(l_4) = \text{time}(l_5) - 1$ while $\text{scope}(l_5) = \text{nchc}.alt(2)\|2$) and $l_9$ is the last location of the 2nd operand (identified using similar reasoning). Hence,

$$\text{vec.map}(l_{10}) = \{v_{l_{10}}^{(1)}, v_{l_{10}}^{(2)}, v_{l_{10}}^{(3)}\} = \{(a1a2, \Lambda, d1), (a1a3, a4, \Lambda), (a1a3, \Lambda, d2)\}$$

The component vector $(a1a2, \Lambda, d1)$ describes the behaviour of component $B$ in case the scenario in the first operand of the enclosing alt is executed. The component vectors $(a1a3, a4, \Lambda)$ and $(a1a3, \Lambda, d2)$ describe behaviour of the component in case the scenario in the second operand of the enclosing alt is executed. Further, the vector $(a1a3, a4, \Lambda)$ describes behaviour of $B$ in case the 1st operand of the nested alt is executed while vector $(a1a3, \Lambda, d2)$ describes behaviour of the component in case the scenario in the 2nd operand of the nested alt is executed.

A more interesting example of translating nested locations into component vectors, which demonstrates the case where an alt fragment is within an operand of a par fragment, is included in Appendix A.
3.2.8 Obtaining the component language

So far we have seen how locations along a lifeline in a sequence diagram can be mapped onto component vectors. We have addressed locations appearing in the main diagram (which amounts to the seq interaction fragment), in an alt, in a par and in a sim interaction fragment. We have also considered nested locations. The corresponding component vectors capture the observable behaviours at each point in the diagram. They provide a snapshot of component behaviour in that they show what events have occurred on the component’s interfaces, starting from the top of the diagram and subsequently moving downwards. The component language \( V \) (of Definition 3.1.3) of a component \( c \) can thus be obtained in a straightforward manner by taking the union of all component vectors associated with a location along the corresponding lifeline. This is done once the final location corresponding to the end of the diagram is reached (i.e. location \( l \in \text{Loc} \) such that \( \text{scope}(l) = \alpha.\text{name} \)).

The only locations that do not adhere to this rationale are locations within a par or a sim interaction fragment. We have already seen that the events associated with these locations are not ordered in any way (including simultaneity). This is captured in the vector mapping of the end location of these two fragments. The vectors associated with the end locations of par and sim capture the observable behaviours of the component while it engages in the interactions described within them and this is what we want to include in the resulting component language. Note that the rest of the locations are also mapped onto component vectors but this is only for recording the occurrence of the associated events, which is necessary for the vector mapping of subsequent locations.

Thus, in obtaining the component language we include all vectors associated with a location along the corresponding lifeline, except for those appearing within a par or a sim fragment, for which we only include the vectors of their end locations. This is formally put in the following definition.

**Definition 3.2.7.** Let \( \Sigma \) be the signature of a component \( c \) represented in a sequence diagram by a lifeline \( \text{Cline} \). Then, its corresponding component language \( V \) is given by

\[
V = \{ \text{vct}_\text{seq}(l) \mid l \in \text{Loc} \setminus (\text{Loc}_{\text{sim}} \cup \text{Loc}_{\text{par}}) \}
\]
This says that the component language $V$ comprises the sets $\text{vec.map}(l)$ for each location $l$ along $\text{Cline}$, providing the location is not within a $\text{par}$ or a $\text{sim}$ interaction fragment.

Based on the postulate of the definition and the formal construction given for $\text{vec.map}$ it may be shown that the resulting set $V$ is a well-defined subset of the set of all possible component vectors $V_\Sigma$, formed over a signature $\Sigma$.

**Proposition 3.2.1.** The set $V$ obtained following the construction given in Definition 3.2.7 is a well-defined subset of $V_\Sigma$.

**Proof.**

By Definition 3.2.7, $V$ comprises sets $\text{vec.map}(l)$, for all $l \in \text{Loc} \setminus (\text{Loc}_{sim} \cup \text{Loc}_{par})$. We need to show that these sets are subsets of $V_\Sigma$. We have seen that

$$\text{Loc} = \text{Loc}^\prime \cup X_1 \cup X_2 \cup Y_1 \cup Y_2$$

We examine each case separately.

- If $l \in \text{Loc}^\prime$, then the vector mapping is given by Definition 3.2.2 in which the function $\text{vec.map}$ is defined from $\text{Loc}^\prime$ to the powerset of $V_\Sigma$. Thus, it maps a location $l \in \text{Loc}^\prime$ onto a subset of $V_\Sigma$.

- If $l \in X_1$, then the vector mapping is given by Definition 3.2.4 in which case the vectors associated with $l$ are given by the union of the vectors returned by $\text{vec.map}$ for the last location from each operand. These vectors are obtained by applying Definition 3.2.2, hence they are component vectors. Thus, $\text{vec.map}(l)$, $l \in X_1$, is a subset of $V_\Sigma$.

- If $l \in Y_1$, then the vector mapping is given by Definition 3.2.3 in which $\text{vec.map}$ of Definition 3.2.2 is used but applied each time to the start location of $\text{alt}$ instead of the previous location. Thus, the vectors returned are component vectors. Hence, $\text{vec.map}(l)$, $l \in Y_1$, is a subset of $V_\Sigma$.

- If $l \in Y_2$, then the vector mapping is given by Definition 3.2.5 in which case we come to the same conclusion using similar reasoning to the case $l \in Y_1$ above.
3.2. From UML to the Component Language

- If \( l \in X_2 \), then the vector mapping is given by Definition 3.2.6 in which case the vectors associated with \( l \) are given by the union of the vectors returned by \textit{vec.map} for each location within the \textit{par} fragment, and for each of the three cases, i.e. \( I, II \) and \( III \). In cases \( I \) and \( II \), the locations are mapped onto vectors using Definition 3.2.2 and Definition 3.2.5 (for the first location of each operand). Thus, in either case the returned vectors are component vectors. In case \( III \), the \( \sqcup \) of two component vectors is computed coordinate-wise and is also a component vector. Hence, the vectors returned by case \( III \) are component vectors. It follows that \textit{vec.map}(\( l \)), \( l \in X_2 \) is a subset of \( V_D \).

Thus, \textit{vec.map}(\( l \)), for all \( l \in \text{Loc} \) is a subset of \( V_D \). It follows that \( V \) of Definition 3.2.7 is a well-defined subset of \( V_D \) as required. □

We have given a formal construction that uses UML2.0 sequence diagrams for obtaining the component language which describes the intended behaviour of the component on its interfaces. In a component language there are vectors which are maximal in the sense that they do not describe earlier behaviour than any other vector in the language.

We characterise maximal component vectors in Chapter 4 where we study the order theoretic properties of a component language.

For now, it suffices to understand that these are precisely the component vectors associated with the location in the sequence diagram which is one before the final location\(^3\) (i.e. the location \( l \in \text{Loc} \) for which \( \text{time}(l) = \text{time}(l_{\text{fin}}) - 1 \), where \( l_{\text{fin}} \in \text{Loc} \) is such that \( \text{scope}(l_{\text{fin}}) = \alpha.\text{name} \)).

The component is not expected to engage only once in the interaction described in a sequence diagram, even if this is an IOD in UML2.0 [OMG04] notation or an hMSC in MSC notation [IT96]. However, it is the case that each time it engages in the interaction it does so from the beginning of the diagram. In other words, the sequence of events

\(^3\)Technically, the component vectors associated with the final location in a sequence diagram are the same as those of its previous location. We have chosen though to associate the final location with obtaining the component language as a whole, by application of Definition 3.2.7, and thus to avoid confusion we consider the component vectors corresponding to its previous location as being the maximal component vectors.
(exchanges) described in a sequence diagram must be completed to the end before the interaction can be repeated. In this sense, maximal component vectors are particularly useful in identifying the point in which the whole interaction can be repeated.

With every repetition of the interaction, the resulting component vectors will be essentially the same as the component vectors of the previous one, with the difference that the sequences on their coordinates will be prefixed by the sequences of the maximal component vectors corresponding to the previous iteration. This capitalises on the fact that an interaction (or scenario) described in a sequence diagram can only be restarted from the beginning, and its execution effectively repeats the sequences of events initially extracted following our formal construction.

In further explanation, the $k$-th iteration will result in component vectors whose sequences are prefixed by the sequences of the maximal component vectors (in $V$) repeated $(k - 1)$ times. This implies that it is appropriate to consider the component language $V$, obtained following the construction given in this chapter, as describing a pattern of behaviour of the component which may be repeated arbitrary many times. Consequently, our analysis and reasoning can be confined within this pattern of behaviour that the component is expected to exhibit whilst making its services available to other components and the environment.

### 3.3 Illustration by example

In this section, we illustrate our approach to obtaining the vector language part of a component from sequence diagrams by means of an example case study from the telecommunications industry. Sequence diagrams, in particular MSCs, are widely used in telecommunications for describing communication protocols [MTJ03, BBJ+02, Mit05].

Figure 3.9 is an anonymised example from a Motorola case study [BBJ+02]. It depicts a UML 2.0 sequence diagram that describes traffic channel allocation and activation between various components for a telecommunication protocol. Notice that communication is asynchronous in this protocol. This is represented by the use of open arrowheads
on messages. Component $A$ has delegated the task of determining what resource to allocate to component $B$.

![Diagram](image)

**Figure 3.9: UML 2.0 case study example**

We are interested in obtaining the component language $V$ for component $B$. The signature of $B$ is given by $\Sigma_B = (P_B, R_B, \beta_B)$ where

- $P_B = \{i_1, i_4, i_6, i_8\}$
- $R_B = \{i_2, i_3, i_5, i_7\}$
- $\beta_B$ is given by
\[ \begin{align*}
\beta_B(i_1) &= \{a1\}, \beta_B(i_2) = \{a2\}, \beta_B(i_3) = \{d1\}, \beta_B(i_4) = \{d2\}, \\
\beta_B(i_5) &= \{e1\}, \beta_B(i_6) = \{e2\}, \beta_B(i_7) = \{f1\}, \beta_B(i_8) = \{f2\}.
\end{align*} \]

We write \((x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)\) for component vectors in \(V_{2n}\) where \(x_k = x(i_k)\), \(k = 1, \ldots, 8\). In other words, the first coordinate of the component vectors corresponds to the interface \(i_1\), the second to \(i_2\) and so on.

\(l_0\) is the initial location and by definition is mapped onto \(A_{B_B}\). Thus,
\[ \text{vec.map}(l_0) = (\Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda) \]

The next location visited is location \(l_1\) since \(\text{time}(l_1) = \text{time}(l_0) + 1\). This location belongs to \(\text{Loc}'\) and by Definition 3.2.2 we have, \(\text{vec.map}(l_1) = v_1\). It is mapped onto a single component vector since \(m\) of the definition is \(m = |\text{vec.map}(l_0)| = 1\) where \(l_0\) is such that \(\text{time}(l_0) = \text{time}(l_1) - 1\). The component vector \(v_1\) is given by
\[ v_1 = (v_{l_1}, v_{l_2}, v_{l_3}, v_{l_4}, v_{l_5}, v_{l_6}, v_{l_7}, v_{l_8}) \]
where each coordinate is given by

- \(v_{l_1} = v_{l_1} = a1 = a1\) since \((a1, l_1) \in RE_B \land a1 \in \beta(i_1)\)
- \(v_{l_2} = v_{l_2} = \Lambda\) since \((a1, l_1) \in RE_B\) but \(a1 \notin \beta_B(i_2)\)
- \(v_{l_3} = v_{l_3} = \Lambda\) since \((a1, l_1) \in RE_B\) but \(a1 \notin \beta_B(i_3)\)
- \(v_{l_4} = v_{l_4} = \Lambda\) since \((a1, l_1) \in RE_B\) but \(a1 \notin \beta_B(i_4)\)
- \(v_{l_5} = v_{l_5} = \Lambda\) since \((a1, l_1) \in RE_B\) but \(a1 \notin \beta_B(i_5)\)
- \(v_{l_6} = v_{l_6} = \Lambda\) since \((a1, l_1) \in RE_B\) but \(a1 \notin \beta_B(i_6)\)
- \(v_{l_7} = v_{l_7} = \Lambda\) since \((a1, l_1) \in RE_B\) but \(a1 \notin \beta_B(i_7)\)
- \(v_{l_8} = v_{l_8} = \Lambda\) since \((a1, l_1) \in RE_B\) but \(a1 \notin \beta_B(i_8)\)

Hence, \(\text{vec.map}(l_1) = (a1, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda)\).

The next location reached is location \(l_2\) (since \(\text{time}(l_2) = \text{time}(l_1) + 1\)). This location marks the beginning of the \text{par} fragment with 3 operands, since \(\text{scope}(l_2) = \)
3.3. Illustration by example

$sdia.par(3)$. Nevertheless, it belongs to $Loc'$ and thus by applying Definition 3.2.2, we have $vec.map(l_2) = y_{l_2}$. It is mapped onto a single component vector since $m$ of the definition is $m = |vec.map(l_1)| = 1$ and $l_1$ is such that $time(l_1) = time(l_2) - 1$. The component vector $y_{l_2}$ is given by

$$y_{l_2} = (v_{l_2_1}, v_{l_2_2}, v_{l_2_3}, v_{l_2_4}, v_{l_2_5}, v_{l_2_6}, v_{l_2_7}, v_{l_2_8})$$

where each coordinate is given by

- $v_{l_2_1} = v_{l_2_2} = a_1$ since no event is associated with location $l_2$
- $v_{l_2_3} = v_{l_2_4} = \Lambda$ since no event is associated with location $l_2$
- $v_{l_2_5} = v_{l_2_6} = \Lambda$ since no event is associated with location $l_2$
- $v_{l_2_7} = v_{l_2_8} = \Lambda$ since no event is associated with location $l_2$

Hence, $vec.map(l_2) = (a_1, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda, \Lambda)$.

The next location considered (not necessarily the next one to be reached, as explained before) is $l_3$. This is the first location of the 1st operand of $par$ since $time(l_3) = time(l_2) + 1 \land scope(l_2) = sdia.par(3)$ while $scope(l_3) = sdia.par(3)\|1$. Thus, its associated component vectors are given by Definition 3.2.3 as follows.

$vec.map(l_3) = y_{l_3}$ since $m = |vec.map(l_2)| = 1$ and $scope(l_2) = sdia.par(3)$. The component vector $y_{l_3}$ is given by

$$y_{l_3} = (v_{l_3_1}, v_{l_3_2}, v_{l_3_3}, v_{l_3_4}, v_{l_3_5}, v_{l_3_6}, v_{l_3_7}, v_{l_3_8})$$

where each coordinate is given by

- $v_{l_3_1} = v_{l_3_2} = a_1$ since $(l_3, d_1) \in SE_B$ but $d_1 \not\in \beta_B(t_1)$
\[
\begin{align*}
\bullet v_{l_3} &= v_{l_2} = \Lambda \text{ since } (l_3, d1) \in SE_B \text{ but } d1 \not\in \beta_B(i_2) \\
\bullet v_{l_3} &= v_{l_2}, d1 = d1 \text{ since } (l_3, d1) \in SE_B \land d1 \in \beta_B(i_3) \\
\bullet v_{l_4} &= v_{l_2} = \Lambda \text{ since } (l_3, d1) \in SE_B \text{ but } d1 \not\in \beta_B(i_4) \\
\bullet v_{l_5} &= v_{l_2} = \Lambda \text{ since } (l_3, d1) \in SE_B \text{ but } d1 \not\in \beta_B(i_5) \\
\bullet v_{l_6} &= v_{l_2} = \Lambda \text{ since } (l_3, d1) \in SE_B \text{ but } d1 \not\in \beta_B(i_6) \\
\bullet v_{l_7} &= v_{l_2} = \Lambda \text{ since } (l_3, d1) \in SE_B \text{ but } d1 \not\in \beta_B(i_7) \\
\bullet v_{l_8} &= v_{l_2} = \Lambda \text{ since } (l_3, d1) \in SE_B \text{ but } d1 \not\in \beta_B(i_8)
\end{align*}
\]

Hence, \( \text{vec.map}(l_3) = \{a1, \Lambda, d1, A, A, A, A, A, A, A\} \).

The next location considered is \( l_4 \) (since \( \text{time}(l_4) = \text{time}(l_3) + 1 \)). This location belongs to \( \text{Loc}' \) and is treated as such by Definition 3.2.2. Thus, we have \( \text{vec.map}(l_4) = v_{l_4} \) since \( m = |\text{vec.map}(l_3)| = 1 \) where \( \text{time}(l_3) = \text{time}(l_4) - 1 \). The component vector \( v_{l_4} \) is given by

\[
v_{l_4} = (v_{l_{t_1}}, v_{l_{t_2}}, v_{l_{t_3}}, v_{l_{t_4}}, v_{l_{t_5}}, v_{l_{t_6}}, v_{l_{t_7}}, v_{l_{t_8}})
\]

where each coordinate is given by

\[
\begin{align*}
\bullet v_{l_4} &= v_{l_3} = a1 \text{ since } (d2, l_4) \in RE_B \text{ but } d2 \not\in \beta_B(i_1) \\
\bullet v_{l_4} &= v_{l_3} = \Lambda \text{ since } (d2, l_4) \in RE_B \text{ but } d2 \not\in \beta_B(i_2) \\
\bullet v_{l_4} &= v_{l_3} = d1 \text{ since } (d2, l_4) \in RE_B \text{ but } d2 \not\in \beta_B(i_3) \\
\bullet v_{l_4} &= v_{l_3}, d2 = d2 \text{ since } (d2, l_4) \in RE_B \land d2 \in \beta_B(i_4) \\
\bullet v_{l_4} &= v_{l_3} = \Lambda \text{ since } (d2, l_4) \in RE_B \text{ but } d2 \not\in \beta_B(i_5) \\
\bullet v_{l_4} &= v_{l_3} = \Lambda \text{ since } (d2, l_4) \in RE_B \text{ but } d2 \not\in \beta_B(i_6) \\
\bullet v_{l_4} &= v_{l_3} = \Lambda \text{ since } (d2, l_4) \in RE_B \text{ but } d2 \not\in \beta_B(i_7) \\
\bullet v_{l_4} &= v_{l_3} = \Lambda \text{ since } (d2, l_4) \in RE_B \text{ but } d2 \not\in \beta_B(i_8)
\end{align*}
\]
Hence, \( \text{vec.map}(l_4) = (a_1, A, d_1, d_2, A, A, A, A) \).

The next location considered (not necessarily the next one to be reached, as explained before) is \( l_5 \). This is the first location of the 2nd operand since \( \text{time}(l_5) = \text{time}(l_4) + 1 \land \text{scope}(l_4) = \text{dia.par}(3) \) whereas \( \text{scope}(l_5) = \text{dia.par}(3) \). This may be expressed more succinctly by saying that \( l_5 \in \mathcal{Y}_2 \). Thus, its associated component vectors are given by Definition 3.2.5 as follows.

\[
\text{vec.map}(l_5) = v_{l_5} \quad \text{since} \quad m = |\text{vec.map}(l_5)| = 1 \quad \text{and} \quad \text{scope}(l_5) = \text{dia.par}(3). \]

The component vector \( v_{l_5} \) is given by

\[
v_{l_5} = (v_{l_51}, v_{l_52}, v_{l_53}, v_{l_54}, v_{l_55}, v_{l_56}, v_{l_57}, v_{l_58})
\]

where each coordinate is given by

- \( v_{l_51} = u_{l_51} = a_1 \) since \( (l_5, e_1) \in \mathcal{SE}_B \) but \( e_1 \not\in B(i_1) \)
- \( v_{l_52} = u_{l_52} = A \) since \( (l_5, e_1) \in \mathcal{SE}_B \) but \( e_1 \not\in B(i_2) \)
- \( v_{l_53} = u_{l_53} = A \) since \( (l_5, e_1) \in \mathcal{SE}_B \) but \( e_1 \not\in B(i_3) \)
- \( v_{l_54} = u_{l_54} = A \) since \( (l_5, e_1) \in \mathcal{SE}_B \) but \( e_1 \not\in B(i_4) \)
- \( v_{l_55} = u_{l_55} = e_1 \) since \( (l_5, e_1) \in \mathcal{SE}_B \) and \( e_1 \in B(i_5) \)
- \( v_{l_56} = u_{l_56} = A \) since \( (l_5, e_1) \in \mathcal{SE}_B \) but \( e_1 \not\in B(i_6) \)
- \( v_{l_57} = u_{l_57} = A \) since \( (l_5, e_1) \in \mathcal{SE}_B \) and \( e_1 \not\in B(i_7) \)
- \( v_{l_58} = u_{l_58} = A \) since \( (l_5, e_1) \in \mathcal{SE}_B \) but \( e_1 \not\in B(i_8) \)

Hence, \( \text{vec.map}(l_5) = (a_1, A, A, A, e_1, A, A, A) \).

The next location considered is \( l_6 \) (since \( \text{time}(l_6) = \text{time}(l_5) + 1 \)). This location belongs to \( \mathcal{L}oc' \) and is treated as such by Definition 3.2.2. Thus, we have

\[
\text{vec.map}(l_6) = v_{l_6} \quad \text{since} \quad m = |\text{vec.map}(l_6)| = 1 \quad \text{where} \quad \text{time}(l_6) = \text{time}(l_5) - 1. \]

The component vector \( v_{l_6} \) is given by

\[
v_{l_6} = (v_{l_61}, v_{l_62}, v_{l_63}, v_{l_64}, v_{l_65}, v_{l_66}, v_{l_67}, v_{l_68})
\]

where each coordinate is given by
• \( v_{i_1} = v_{i_2} = a_1 \) since \((e_2, l_6) \in RE_B\) but \(e_2 \notin \beta_B(i_1)\)
• \( v_{i_5} = v_{i_6} = a_1 \) since \((e_2, l_6) \in RE_B\) but \(e_2 \notin \beta_B(i_2)\)
• \( v_{i_6} = v_{i_3} = a_1 \) since \((e_2, l_6) \in RE_B\) but \(e_2 \notin \beta_B(i_3)\)
• \( v_{i_6} = v_{i_4} = a_1 \) since \((e_2, l_6) \in RE_B\) but \(e_2 \notin \beta_B(i_4)\)
• \( v_{i_6} = v_{i_5} = e_1 \) since \((e_2, l_6) \in RE_B\) but \(e_2 \notin \beta_B(i_5)\)
• \( v_{i_7} = v_{i_6} = e_2 \) since \((e_2, l_6) \in RE_B \land e_2 \in \beta_B(i_6)\)
• \( v_{i_7} = v_{i_7} = a_1 \) since \((e_2, l_6) \in RE_B\) but \(e_2 \notin \beta_B(i_7)\)
• \( v_{i_8} = v_{i_8} = a_1 \) since \((e_2, l_6) \in RE_B\) but \(e_2 \notin \beta_B(i_8)\)

Hence, \( \text{vec.map}(l_6) = (a_1, a_1, a_1, a_1, e_1, e_2, a_1) \).

The next location considered (not necessarily the next one to be reached, as explained before) is \( l_7 \). This is the first location of the 3rd operand since \( \text{time}(l_7) = \text{time}(l_6) + 1 \land \text{scope}(l_6) = \text{sdia.par}(3) \# 2 \) whereas \( \text{scope}(l_7) = \text{sdia.par}(3) \# 3 \). This may be expressed more succinctly by saying that \( l_7 \in Y_2 \). Thus, its associated component vectors are given by Definition 3.2.5 as follows.

\[
\text{vec.map}(l_7) = v_{l_7} \text{ since } m = |\text{vec.map}(l_2)| = 1 \text{ and } \text{scope}(l_2) = \text{sdia.par}(3). \text{ The component vector } v_{l_7} \text{ is given by }
\]

\[
v_{l_7} = (v_{l_{i_1}}, v_{l_{i_2}}, v_{l_{i_3}}, v_{l_{i_4}}, v_{l_{i_5}}, v_{l_{i_6}}, v_{l_{i_7}}, v_{l_{i_8}})
\]

where each coordinate is given by

• \( v_{l_{i_1}} = v_{l_{i_2}} = a_1 \) since \((l_7, f_1) \in SE_B\) but \(f_1 \notin \beta_B(i_1)\)
• \( v_{l_{i_2}} = v_{l_{i_2}} = a_1 \) since \((l_7, f_1) \in SE_B\) but \(f_1 \notin \beta_B(i_2)\)
• \( v_{l_{i_3}} = v_{l_{i_2}} = a_1 \) since \((l_7, f_1) \in SE_B\) but \(f_1 \notin \beta_B(i_3)\)
• \( v_{l_{i_4}} = v_{l_{i_2}} = a_1 \) since \((l_7, f_1) \in SE_B\) but \(f_1 \notin \beta_B(i_4)\)
• \( v_{l_{i_5}} = v_{l_{i_2}} = a_1 \) since \((l_7, f_1) \in SE_B\) but \(f_1 \notin \beta_B(i_5)\)
3.3. Illustration by example

• \( v_{l_0} = v_{l_9} = \Lambda \) since \((l_7, f1) \in SE_B \) but \( f1 \not\in \beta_B(i_8) \)

• \( v_{l_{17}} = v_{l_{27}} \cdot f1 = f1 \) since \((l_7, f1) \in SE_B \land f1 \in \beta_B(i_{17}) \)

• \( v_{l_{26}} = v_{l_{25}} = \Lambda \) since \((l_7, f1) \in SE_B \) but \( f1 \not\in \beta_B(i_8) \)

Hence, \( vec\_map(l_7) = (a_1, \Lambda, \Lambda, \Lambda, \Lambda, A, f1, A) \).

The next location considered is \( l_8 \) (since \( time(l_8) = time(l_7) + 1 \)). This location belongs to \( Loc' \) and is treated as such by Definition 3.2.2. Thus, we have

\[
vec\_map(l_8) = v_{l_8} \quad \text{since} \quad m = |vec\_map(l_7)| = 1 \quad \text{where} \quad time(l_7) = time(l_8) - 1.
\]

The component vector \( v_{l_8} \) is given by

\[
v_{l_8} = (v_{l_{81}}, v_{l_{82}}, v_{l_{83}}, v_{l_{84}}, v_{l_{85}}, v_{l_{86}}, v_{l_{87}}, v_{l_{88}})
\]

where each coordinate is given by

• \( v_{l_{81}} = v_{l_{21}} = a_1 \) since \((f2, l_8) \in RE_B \) but \( f2 \not\in \beta_B(i_1) \)

• \( v_{l_{82}} = v_{l_{22}} = \Lambda \) since \((f2, l_8) \in RE_B \) but \( f2 \not\in \beta_B(i_2) \)

• \( v_{l_{83}} = v_{l_{23}} = \Lambda \) since \((f2, l_8) \in RE_B \) but \( f2 \not\in \beta_B(i_3) \)

• \( v_{l_{84}} = v_{l_{24}} = \Lambda \) since \((f2, l_8) \in RE_B \) but \( f2 \not\in \beta_B(i_4) \)

• \( v_{l_{85}} = v_{l_{25}} = \Lambda \) since \((f2, l_8) \in RE_B \) but \( f2 \not\in \beta_B(i_5) \)

• \( v_{l_{86}} = v_{l_{26}} = \Lambda \) since \((f2, l_8) \in RE_B \) but \( f2 \not\in \beta_B(i_6) \)

• \( v_{l_{87}} = v_{l_{27}} \cdot f1 = f1 \) since \((f2, l_8) \in RE_B \land f2 \not\in \beta_B(i_{17}) \)

• \( v_{l_{88}} = v_{l_{28}} \cdot f2 = f2 \) since \((f2, l_8) \in RE_B \land f2 \not\in \beta_B(i_8) \)

Hence, \( vec\_map(l_8) = (a_1, A, A, A, A, A, A, A, f1, f2) \).

The next location considered is location \( l_9 \). This is the end location of the \text{par} fragment, i.e., \( l_9 \in X_2 \), since \( \text{scope}(l_9) = \text{sdia}\_\text{par}(3) \). Thus, the component vectors associated with it are given by Definition 3.2.6 as follows.
First, we determine the three cases $\text{vec.map}(l)^I$, $\text{vec.map}(l)^{II}$ and $\text{vec.map}(l)^{III}$ described in Section 3.2.5, for each location $l$ appearing within the first two operands of $\text{par}$. These are $l_5, l_4, l_5$ and $l_5$.

We start by determining the three cases for location $l_3$, that is, $\text{vec.map}(l_3)^I$, $\text{vec.map}(l_3)^{II}$ and $\text{vec.map}(l_3)^{III}$:

- $\text{vec.map}(l_3)^I = \text{vec.map}(l_3) = (a_1, A, d_1, A, A, A, A, A)$

- $\text{vec.map}(l_3)^{II} = \text{vec.map}(l_5) \cup \text{vec.map}(l_6) \cup \text{vec.map}(l_7) \cup \text{vec.map}(l_8)$
  since $l_5, l_6, l_7, l_8$ are the locations appearing in the succeeding operands which, in the case of the sequence diagram of Figure 3.9, are the 2nd and 3rd operand of the $\text{par}$ fragment. Thus, we have

\[ \text{vec.map}(l_3)^{II} = \text{vec.map}(l_5) \cup \text{vec.map}(l_6) \cup \text{vec.map}(l_7) \cup \text{vec.map}(l_8) \]

\[ = \{(a_1, A, A, A, e_1, A, A, A), (a_1, A, A, A, e_1, e_2, A, A),
       (a_1, A, A, A, A, f_1, A), (a_1, A, A, A, A, f_1, f_2)\} \]

- $\text{vec.map}(l_3)^{III} = \mathcal{V}_{l_5,l_6} \cup \mathcal{V}_{l_5,l_7} \cup \mathcal{V}_{l_5,l_8}$
  since $m = |\text{vec.map}(l_2)| = 1$ where $l_2$ is the start location of $\text{par}$, and $l_5, l_6, l_7, l_8$ are the locations of the succeeding operands of the $\text{par}$ fragment, excluding the last operand. The corresponding component vectors are given by,

\[- \mathcal{V}_{l_5,l_6} = \mathcal{V}_{l_5} \cup \mathcal{V}_{l_6} = (a_1, A, d_1, A, A, A, A, A) \cup (a_1, A, A, A, A, A, A, A)
             \text{Hence, } \mathcal{V}_{l_5,l_6} = (a_1, A, d_1, A, A, A, A, A)\]

\[- \mathcal{V}_{l_5,l_7} = \mathcal{V}_{l_5} \cup \mathcal{V}_{l_7} = (a_1, A, d_1, A, A, A, A, A) \cup (a_1, A, A, A, A, e_1, A, A)
             \text{Hence, } \mathcal{V}_{l_5,l_7} = (a_1, A, d_1, A, e_1, A, A, A)\]

\[- \mathcal{V}_{l_5,l_8} = \mathcal{V}_{l_5} \cup \mathcal{V}_{l_8} = (a_1, A, d_1, A, A, A, A, A) \cup (a_1, A, A, A, A, A, f_1, A)
             \text{Hence, } \mathcal{V}_{l_5,l_8} = (a_1, A, d_1, A, A, A, f_1, A)\]

\[- \mathcal{V}_{l_5,l_9} = \mathcal{V}_{l_5} \cup \mathcal{V}_{l_9} = (a_1, A, d_1, A, A, A, A, A) \cup (a_1, A, A, A, A, A, f_1, f_2)
             \text{Hence, } \mathcal{V}_{l_5,l_9} = (a_1, A, d_1, A, A, A, f_1, f_2)\]

Hence,

\[ \text{vec.map}(l_3)^{III} = \{(a_1, A, d_1, A, e_1, A, A, A), (a_1, A, d_1, A, e_1, A, A, A), \} \]

3.3. Illustration by example

\((a_1, A, d_1, A, A, A, f_1, A), (a_1, A, d_1, A, A, A, f_1, f_2))\)

Next we determine the three cases for location \(l_4\), that is, \(\text{vec.map}(l_4)^I\), \(\text{vec.map}(l_4)^{II}\) and \(\text{vec.map}(l_4)^{III}\).

- \(\text{vec.map}(l_4)^I = \text{vec.map}(l_4) = (a_1, A, d_1, d_2, A, A, A, A)\)

- \(\text{vec.map}(l_4)^{II} = \text{vec.map}(l_5) \cup \text{vec.map}(l_6) \cup \text{vec.map}(l_7) \cup \text{vec.map}(l_8)\)
  
since \(l_5, l_6, l_7, l_8\) are the locations appearing in the succeeding operands, which in the case of the sequence diagram of Figure 3.9 are the 2nd and 3rd operand of the \text{par} fragment. Thus, we have

\[
\begin{align*}
\text{vec.map}(l_4)^{II} &= \text{vec.map}(l_5) \cup \text{vec.map}(l_6) \cup \text{vec.map}(l_7) \cup \text{vec.map}(l_8) \\
&= \{(a_1, A, A, A, e_1, A, A, A), (a_1, A, A, A, e_1, e_2, A, A), \\
&\quad (a_1, A, A, A, A, A, f_1, A), (a_1, A, A, A, A, A, f_1, f_2)\}
\end{align*}
\]

Notice that the set \(\text{vec.map}(l_4)^{II}\) is exactly the same as the set \(\text{vec.map}(l_3)^{II}\). This is indeed the case, because they are locations of the same operand (the first) of \text{par}.

- \(\text{vec.map}(l_4)^{III} = \gamma_{l_4,l_5} \cup \gamma_{l_4,l_6} \cup \gamma_{l_4,l_7} \cup \gamma_{l_4,l_8}\)
  
since \(m = |\text{vec.map}(l_3)| = 1\) where \(l_2\) is the start location of \text{par}, and \(l_5, l_6, l_7, l_8\) are the locations of the succeeding operands of the \text{par} fragment. The corresponding component vectors are obtained in exactly the same way as done for location \(l_3\) above, and we get

\[
\begin{align*}
- \gamma_{l_4,l_5} = \gamma_{l_4} \cup \gamma_{l_5} &= (a_1, A, d_1, d_2, A, A, A, A) \cup (a_1, A, A, A, e_1, A, A, A) \\
\text{Hence, } \gamma_{l_4,l_5} &= (a_1, A, A, d_1, d_2, e_1, A, A, A) \\
- \gamma_{l_4,l_6} = \gamma_{l_4} \cup \gamma_{l_6} &= (a_1, A, d_1, d_2, A, A, A, A) \cup (a_1, A, A, A, e_1, e_2, A, A) \\
\text{Hence, } \gamma_{l_4,l_6} &= (a_1, A, A, d_1, d_2, e_1, e_2, A, A) \\
- \gamma_{l_4,l_7} = \gamma_{l_4} \cup \gamma_{l_7} &= (a_1, A, d_1, d_2, A, A, A, A) \cup (a_1, A, A, A, A, A, f_1, A) \\
\text{Hence, } \gamma_{l_4,l_7} &= (a_1, A, A, d_1, d_2, A, A, f_1, A)
\end{align*}
\]
Chapter 3. A Formal Language for Components

\[ \text{vec.map}(l_4)^{III} = \{(a_1, A, A, d_1, d_2, A, A, A, A), (a_1, A, A, A, d_1, d_2, A, A, A), (a_1, A, A, A, A, d_1, d_2, A, A, A, A, A, A, A, A, f_1, f_2)\} \]

Thus,

\[ \text{vec.map}(l_4)^{III} = \{(a_1, A, d_1, d_2, A, A, A, A, A, f_1, f_2)\} \]

We continue by determining the three cases for location \( l_5 \), that is, \( \text{vec.map}(l_5)^I \), \( \text{vec.map}(l_5)^{II} \) and \( \text{vec.map}(l_5)^{III} \).

- \( \text{vec.map}(l_5)^I = \text{vec.map}(l_5) = (a_1, A, A, A, A, A, A, A) \)
- \( \text{vec.map}(l_5)^{II} = \text{vec.map}(l_7) \cup \text{vec.map}(l_8) \) since \( l_5 \) is a location of the 2nd operand and thus locations \( l_7, l_8 \) are the only locations appearing in the succeeding operands which, in the case of the sequence diagram of Figure 3.9, is only the 3rd operand of the \( \text{par} \) fragment. Thus, we have

\[ \text{vec.map}(l_5)^{II} = \text{vec.map}(l_7) \cup \text{vec.map}(l_8) \]

\[ = \{(a_1, A, A, A, A, A, A, A, f_1, f_2)\} \]

- \( \text{vec.map}(l_5)^{III} = \text{vec.map}(l_2) \cup \text{vec.map}(l_3) \)

since \( m = |\text{vec.map}(l_2)| = 1 \) where \( l_2 \) is the start location of \( \text{par} \), and \( l_7, l_8 \) are the locations of the succeeding operands of the \( \text{par} \) fragment. The corresponding component vectors are given by,

\[ \text{vec.map}(l_5)^{III} = \{(a_1, A, A, A, A, A, A, A, f_1, f_2)\} \]

Hence,

\[ \text{vec.map}(l_6)^{I} = \{(a_1, A, A, A, A, e_1, A, f_1, f_2)\} \]

Hence,

\[ \text{vec.map}(l_6)^{III} = \{(a_1, A, A, A, e_1, A, f_1, f_2)\} \]
We continue by determining the three cases for location $l_6$, that is, $\text{vec.map}(l_6)^I$, $\text{vec.map}(l_6)^{II}$ and $\text{vec.map}(l_6)^{III}$.

- $\text{vec.map}(l_6)^I = \text{vec.map}(l_6) = (a_1, A, A, A, e_1, e_2, A, A)$

- $\text{vec.map}(l_6)^{II} = \text{vec.map}(l_7) \cup \text{vec.map}(l_8)$
  
  since $l_7, l_8$ are the locations appearing in the succeeding operands which, in the case of the sequence diagram of Figure 3.9, is only the 3rd operand of the par fragment. Thus, we have

\[
\text{vec.map}(l_6)^{II} = \text{vec.map}(l_7) \cup \text{vec.map}(l_8) = \{(a_1, A, A, A, A, A, f_1, A), (a_1, A, A, A, A, A, f_1, f_2)\}
\]

Notice that the set $\text{vec.map}(l_6)^{II}$ is exactly the same as the set $\text{vec.map}(l_6)^{III}$. This is indeed the case, because they are locations of the same operand (the second) of par.

- $\text{vec.map}(l_6)^{III} = \psi_{l_6,l_7} \cup \psi_{l_6,l_8}$
  
  since $m = |\text{vec.map}(l_6)| = 1$ where $l_2$ is the start location of par, and $l_7, l_8$ are the locations of the succeeding operands, only the 3rd operand here, of the par fragment. The corresponding component vectors are given by,

\[
\begin{align*}
\psi_{l_6,l_7} &= \psi_{l_6} \cup \psi_{l_7} = (a_1, A, A, A, e_1, e_2, A, A) \cup (a_1, A, A, A, A, A, f_1, A) \\
&= (a_1, A, A, A, e_1, e_2, f_1, A) \\
&\text{Hence, } \psi_{l_6,l_7} = (a_1, A, A, A, e_1, e_2, f_1, A) \\
\psi_{l_6,l_8} &= \psi_{l_6} \cup \psi_{l_8} = (a_1, A, A, A, e_1, e_2, A, A) \cup (a_1, A, A, A, A, A, f_1, f_2) \\
&= (a_1, A, A, A, e_1, e_2, f_1, f_2) \\
&\text{Hence, } \psi_{l_6,l_8} = (a_1, A, A, A, e_1, e_2, f_1, f_2) \\
\end{align*}
\]

Hence,

\[
\text{vec.map}(l_6)^{III} = \{(a_1, A, A, A, e_1, e_2, f_1, A), (a_1, A, A, A, e_1, e_2, f_1, f_2)\}
\]

As explained in Section 3.2.5, there is no need to determine the three cases for locations of the last operand, i.e. $l_7$ and $l_8$. 


Therefore, the next location considered is location $l_9$. This is the end location of the \texttt{par} fragment and therefore its associated component vectors are obtained by applying Definition 3.2.6. Thus,

$$vec\_map(l_9) = vec\_map(l_3)^x \cup vec\_map(l_4)^x \cup vec\_map(l_5)^x \cup vec\_map(l_6)^x$$

where $x$ runs through the three cases for each location, i.e. $x = I, II, III$. Hence,

$$vec\_map(l_9) = \{v_{l_9}^{(1)}, v_{l_9}^{(2)}, ..., v_{l_9}^{(17)}\}$$

$$= \{(a_1, A, d_1, A, A, A, A, A), (a_1, A, A, A, A, e_1, A, A, A), (a_1, A, A, A, A, f_1, A, A, A), (a_1, A, d_1, A, e_1, A, A, A), (a_1, A, d_1, A, e_2, A, A, A), (a_1, A, d_1, A, A, A, A, A), (a_1, A, d_1, A, A, A, f_1, A, A, A), (a_1, A, d_1, A, A, A, f_1, A, A, A), (a_1, A, d_1, A, A, A, f_1, A, A, A), (a_1, A, d_1, d_2, A, A, A, A, A), (a_1, A, d_1, d_2, A, e_1, A, A, A), (a_1, A, d_1, e_1, e_2, A, A, A), (a_1, A, d_1, e_2, f_1, A, A, A), (a_1, A, e_1, A, e_2, A, A, A), (a_1, A, e_1, A, e_2, f_1, A, A, A), (a_1, A, e_1, A, e_2, f_1, A, A, A), (a_1, A, e_1, A, e_2, f_1, A, A, A), (a_1, A, e_1, A, e_2, f_1, A, A, A), (a_1, A, e_1, A, e_2, f_1, A, A, A),$$

The next location to be reached is $l_{10}$ (since $time(l_{10}) = time(l_9) + 1$). This location belongs to $Loc'$ and thus the corresponding component vectors are given by Definition 3.2.2 as follows.

$$vec\_map(l_{10}) = \{v_{l_{10}}^{(1)}, v_{l_{10}}^{(2)}, ..., v_{l_{10}}^{(17)}\}$$

since $m = |vec\_map(l_9)| = 17$ and $l_9$ is the location such that $time(l_9) = time(l_{10})$.

The component vector $v_{l_{10}}^{(1)}$ is given by

$$v_{l_{10}}^{(1)} = (v_{l_{10}}^{(1)}_{101}, v_{l_{10}}^{(1)}_{102}, v_{l_{10}}^{(1)}_{103}, v_{l_{10}}^{(1)}_{104}, v_{l_{10}}^{(1)}_{105}, v_{l_{10}}^{(1)}_{106}, v_{l_{10}}^{(1)}_{107}, v_{l_{10}}^{(1)}_{108})$$

where each coordinate is given by

- $v_{l_{10}}^{(1)}_{101} = v_{l_9}^{(1)}_{101} = a_1$ since $(l_{10}, a_2) \in SE_B$ but $a_2 \notin \beta_B(v_1)$
3.3. Illustration by example

\[ v^{(1)}_{i_{10}} = v^{(1)}_{i_{10}} a_2 = a_2 \text{ since } (l_{10}, a_2) \in SE_B \wedge a_2 \in \beta_B(i_2) \]

\[ v^{(1)}_{i_{102}} = v^{(1)}_{i_{102}} d_1 \text{ since } (l_{10}, a_2) \in SE_B \text{ but } a_2 \notin \beta_B(i_3) \]

\[ v^{(1)}_{i_{104}} = v^{(1)}_{i_{104}} = a_2 \text{ since } (l_{10}, a_2) \in SE_B \wedge a_2 \in \beta_B(i_4) \]

\[ v^{(1)}_{i_{105}} = v^{(1)}_{i_{105}} = a_2 \text{ since } (l_{10}, a_2) \in SE_B \text{ but } a_2 \notin \beta_B(i_5) \]

\[ v^{(1)}_{i_{106}} = v^{(1)}_{i_{106}} = a_2 \text{ since } (l_{10}, a_2) \in SE_B \text{ but } a_2 \notin \beta_B(i_6) \]

\[ v^{(1)}_{i_{107}} = v^{(1)}_{i_{107}} = a_2 \text{ since } (l_{10}, a_2) \in SE_B \text{ but } a_2 \notin \beta_B(i_7) \]

\[ v^{(1)}_{i_{108}} = v^{(1)}_{i_{108}} = a_2 \text{ since } (l_{10}, a_2) \in SE_B \text{ but } a_2 \notin \beta_B(i_8) \]

Hence,

\[ \psi_{(1)}^{(1)} = (v_{i_{10}}, v_{i_{102}}, v_{i_{104}}, v_{i_{105}}, v_{i_{106}}, v_{i_{107}}, v_{i_{108}}) = (a_1, a_2, d_1, a_2, a_2, a_2, a_2) \]

The rest of the component vectors associated with location \( l_{10} \) are computed in similar fashion. Here, we have only included the case for \( \psi_{(1)}^{(2)} \).

The component vector \( \psi_{(1)}^{(2)} \) is given by

\[ \psi_{(1)}^{(2)} = (v_{i_{10}}, v_{i_{102}}, v_{i_{104}}, v_{i_{105}}, v_{i_{106}}, v_{i_{107}}, v_{i_{108}}) \]

where each coordinate is given by

\[ v_{i_{10}}^{(2)} = v_{i_{10}}^{(2)} = a_1 \text{ since } (l_{10}, a_2) \in SE_B \text{ but } a_2 \notin \beta_B(i_1) \]

\[ v_{i_{102}}^{(2)} = v_{i_{102}}^{(2)} = d_1 \text{ since } (l_{10}, a_2) \in SE_B \wedge a_2 \in \beta_B(i_2) \]

\[ v_{i_{104}}^{(2)} = v_{i_{104}}^{(2)} = a_2 \text{ since } (l_{10}, a_2) \in SE_B \text{ but } a_2 \notin \beta_B(i_3) \]

\[ v_{i_{105}}^{(2)} = v_{i_{105}}^{(2)} = a_2 \text{ since } (l_{10}, a_2) \in SE_B \wedge a_2 \in \beta_B(i_4) \]

\[ v_{i_{106}}^{(2)} = v_{i_{106}}^{(2)} = a_2 \text{ since } (l_{10}, a_2) \in SE_B \text{ but } a_2 \notin \beta_B(i_5) \]

\[ v_{i_{107}}^{(2)} = v_{i_{107}}^{(2)} = a_2 \text{ since } (l_{10}, a_2) \in SE_B \text{ but } a_2 \notin \beta_B(i_6) \]

\[ v_{i_{108}}^{(2)} = v_{i_{108}}^{(2)} = a_2 \text{ since } (l_{10}, a_2) \in SE_B \text{ but } a_2 \notin \beta_B(i_8) \]
Hence,

$$y_{l_{10}}^{(2)} = (v_{l_{10}}^{(2)}, v_{l_{10}}^{(2)}, v_{l_{10}}^{(2)}, v_{l_{10}}^{(2)}, v_{l_{10}}^{(2)}, v_{l_{10}}^{(2)}, v_{l_{10}}^{(2)}, v_{l_{10}}^{(2)} = (a_1, a_2, A, A, e_1, A, A, A)$$

Following the same process for the rest of the vectors, location $l_{10}$ is mapped onto the following set of component vectors.

$$\text{vec.map}(l_{10}) = \{y_{l_{10}}^{(1)}, y_{l_{10}}^{(2)}, \ldots, y_{l_{10}}^{(17)}\}$$

$$= \{(a_1, a_2, d_1, A, A, A, A, A), (a_1, a_2, A, A, A, A, A, f_1, A), (a_1, a_2, A, A, A, A, f_1, f_2),$$
$$ (a_1, a_2, d_1, A, e_1, A, A, A), (a_1, a_2, d_1, A, e_1, e_2, A, A),$$
$$ (a_1, a_2, d_1, A, A, A, f_1, A), (a_1, a_2, d_1, A, A, A, f_1, f_2),$$
$$ (a_1, a_2, d_1, d_2, A, A, A, A), (a_1, a_2, d_1, d_2, e_1, A, A, A),$$
$$ (a_1, a_2, d_1, d_2, e_1, e_2, A, A), (a_1, a_2, d_1, d_2, A, A, A, f_1, A),$$
$$ (a_1, a_2, d_1, d_2, A, A, f_1, f_2), (a_1, a_2, A, A, e_1, A, A, f_1, A),$$
$$ (a_1, a_2, A, A, e_1, A, e_2, f_1, A), (a_1, a_2, A, A, e_1, e_2, f_1, A),$$

Finally, we may now obtain the component language of component $B$. By applying Definition 3.2.7 as described in Section 3.2 we have,

$$V_B = \text{vec.map}(l_0) \cup \text{vec.map}(l_1) \cup \text{vec.map}(l_2) \cup \text{vec.map}(l_8) \cup \text{vec.map}(l_{10})$$

$$ (a_1, A, d_1, A, A, A, A, A, A), (a_1, A, A, A, A, A, e_1, A, A, A),$$
$$ (a_1, A, A, A, A, A, f_1, A), (a_1, A, A, A, A, A, f_1, f_2),$$
$$ (a_1, A, d_1, A, e_1, A, A, A), (a_1, A, d_1, A, e_1, e_2, A, A),$$
$$ (a_1, A, d_1, A, A, A, f_1, A), (a_1, A, d_1, A, A, A, f_1, f_2),$$
$$ (a_1, A, d_1, d_2, A, A, A, A), (a_1, A, d_1, d_2, e_1, A, A, A),$$
$$ (a_1, A, d_1, d_2, e_1, e_2, A, A), (a_1, A, d_1, d_2, A, A, A, f_1, A),$$
$$ (a_1, A, d_1, d_2, A, A, A, f_1, f_2), (a_1, A, A, A, e_1, A, A, f_1, A),$$
The formal description of the component $B$ is thus given by the pair $(\Sigma_B, V_B)$ where $\Sigma_B$ describes the structure of the component in terms of its provided and required interfaces together with the set of events associated with each, while its language $V_B$ comprises component vectors which describe the observable behaviours of the component on its interfaces within the context of its scenario-based specification given in terms of the UML sequence diagram of Figure 3.9.

3.4 Concluding note

We have seen that a component in our approach has multiple provided interfaces through which it offers its services and multiple required interfaces through which it requires services from other components. This is in line with the view of components taken in mainstream software engineering approaches such as UML [OMG04] and also Koala [vOvdLKM00].

In this chapter, we have presented a formal model of components in which a component is associated with a) a signature, which describes its interfaces and b) a language defined over this signature, which describes its intended behaviour. The component signature defines the interfaces the component provides and requires as well as their associated operations. The language part of a component comprises vectors where each coordinate corresponds to an interface and contains a finite sequence of operation calls (more generally, events) that may be experienced on that interface.

This vector language-based description of component behaviour allows to consider concurrency at the level of individual components. We can model concurrent and simultaneous event occurrences on interfaces of the same component. An underlying assumption of this model is that events occur sequentially on a single interface (one at a time). Concurrent and simultaneous event occurrences can only engage distinct interfaces of the component.
Naturally, in component-based development we are interested in the intended or allowed sequences of operation calls on component interfaces. As a result, our goal is to restrict to an appropriate subset of component vectors which describe the intended behaviour only. This subset comprises the corresponding component language.

Among various options for obtaining the component language, we opted for using a scenario-based specification since this underlines pragmatic and industrially accepted approaches to software specification. We first described the idea of obtaining a component language from scenarios in [MS04a]. The use of LSCs for describing component behaviour was proposed in [Mos04] and this included an early attempt to translate scenarios into component vectors.

The approach we advocate here is concerned with scenario-based specifications given in terms of UML2.0 sequence diagrams. We considered their basic features such as sequencing, alternative and parallel interaction fragments. We also addressed the case of an interaction fragment enclosing another interaction fragment within one of its operands. These features also appear in MSCs and LSCs and hence our approach applies equally well across scenario description languages (SDLs).

This part of the UML notation, concerned with scenario-based specification within the modelling language, was given a more concrete semantics in terms of a vector language. We proposed the use of an additional interaction fragment for capturing simultaneous event occurrences. Simultaneity was considered, as one possible case, in the semantics of the parallel interaction fragment par. The semantics of the sim fragment though is different in that it says the associated events have to occur simultaneously. The difference between events appearing in a par and a sim fragment can be understood as the distinction between may and must happen concurrently.

The component language was obtained by mapping each point in the diagram onto (a set of) component vectors. The idea is that the corresponding component vectors describe the observable behaviours of the component during the interaction given in the sequence diagram. The collection of these 'snapshots' of behaviour describes the complete behaviour of the component as the first is taken at the top of the diagram and the rest whilst moving downwards.
3.4. Concluding note

Of course, a component will engage a (possibly infinite) number of times in the interaction (or scenario) described in a sequence diagram. Such repetitions concern the interaction as a whole, hence they can only occur when the end of the diagram has been reached and can only start at the beginning of the diagram. This means that it is appropriate to view the obtained component language \( V \) as describing a *pattern of behaviour* that can be repeated arbitrarily many times.

The benefit of using UML 2.0 sequence diagrams for obtaining the component language is that it brings our formal model closer to pragmatic and industrially accepted engineering practices. In the same context, the fact that our model comes equipped with extra information about component interfaces, in terms of the component’s observable behaviours, offers interesting perspectives with respect to system verification. We will see how the additional information about interfaces in a component language can be used to uncover inconsistencies in scenario-based specifications in the following chapter (Chapter 4).

The idea of using a language of vectors to describe component behaviour originates in [SP01] where components are understood as abstract boxes with input and output communication points. It was extended here to address components as software entities with provided and required interfaces, as given by the corresponding signature (Definition 3.1.1), to reflect the contractual use of components in UML and more conventional approaches to component-based software engineering discussed in Chapter 2. Nevertheless, the formal translation of scenarios into component vectors described in this chapter, perhaps with moderate adjustments such as the unique operation names on interfaces, can be seen as a specification language on top of the vector language considered in [SP01].

Finally, to anticipate, the need for keeping all vectors in the resulting component language (instead of keeping only the ones corresponding to the final location of the sequence diagram) stems from the fact that the set of component vectors can be turned into a partially ordered set, where the ordering relation is given by coordinate-wise prefix ordering. It is this order structure of the component language that allows us to determine the ordering relation between events on different interfaces of a component.
Note that a component vector in isolation specifies an ordering among events on the same interface only. For the ordering relation between events occurring on different interfaces, we need the rest of the component vectors too. The order structure and related properties of a component language are discussed in the next chapter.
Chapter 4

Properties of a Component Language

We have seen that the formal description of a component in our approach consists of a signature \( \Sigma \) and a component language \( V \) which is an appropriate subset of all possible component vectors formed over \( \Sigma \). The subset \( V \) for a component was determined using UML2.0 sequence diagrams. This description of a component contains additional information about the observable behaviours on component interfaces.

In this chapter, we examine the order structure of a component language as a means of determining the ordering between events occurring on different interfaces of the component. This is based on the order theoretic properties of its component vectors. A component language can be equipped with an ordering relation, given by coordinate-wise prefix ordering of component vectors, which turns it into a partially ordered set. We describe how sequential, mutually exclusive, concurrent and simultaneous event occurrences on component interfaces manifest themselves in the order structure of the corresponding component language.

By exploiting this order structure, we identify properties of a component language which allow us to determine a particular class of components, the so-called well-behaved components. Based on consequences of well-behavedness, we may relate our component model to a more general theory of non-interleaving representation of behaviour [Shi97].
This lays the foundations for a behavioural model including both an event-oriented and a state-oriented description of a component, which is discussed in subsequent chapters.

Furthermore, in checking a component language against these properties we may uncover potential instances of pathological behaviour. That is, behaviour not intended but emerging from the complex interplay between calls to operations on component interfaces.

We demonstrate these ideas by means of an extended example taken from the consumer electronics industry.

It should be noted that the properties that lead to the characterisation of well-behaved components take up on ideas found in M. W. Shields' study [Shi88, Shi97] of the order theoretic properties of vector languages formed by behaviour vectors. These vectors are closely related to component vectors, as described in Section 3.1.2. In [MSKF03] we described the adaptation of this work in the more general setting provided by component languages and we were also concerned with applying the formalism in obtaining a description of a component in terms of its observable behaviours. The relevance of these order theoretic properties in identifying missing behaviours in component-based design was highlighted in performing the case study appearing in [MSKF03]. In this chapter we outline these developments to the theory and stress its application to component-based software design. Further, we show that missing behaviours in the initial specification may refer to race conditions, a common semantic inconsistency in scenario-based notations and UML sequence diagrams, which were used in Chapter 3 for obtaining component languages.

4.1 Order Theoretic Properties

In this section we present the basic properties of component vectors in terms of prefix ordering, greatest lower and least upper bounds and a right-cancellation operator. We have seen that component vectors are essentially tuples of sequences. We may thus define operations on component vectors in terms of well-known operations on sequences.

If $x$ and $y$ are sequences we write $x.y$ for the concatenation of $x$ and $y$. As is well known,
4.1. Order Theoretic Properties

this operation is associative with identity $A$, where $A$ denotes the empty sequence. We also have a partial order on sequences given by $x \leq y$ if and only if there exists $z$ such that $x.z = y$ ($x$ can be extended to $y$), and this partial order has a bottom element $A$. It is also well known that concatenation is cancellative, so that if $x \leq y$ then the sequence $z$ such that $x.z = y$ is unique. We denote it by $y/x$. In what follows, we lift these concepts to component vectors.

**Definition 4.1.1.** Let $c = (\Sigma, \nu)$ be a component and let $u, v$ be component vectors in $V \subseteq V_{\Sigma}$. We define,

- $u \cdot v$ to be the unique vector $w$ such that $w(i) = u(i) \cdot v(i)$, for each $i \in I_{\Sigma}$ (concatenation)

- $u \leq v$ if and only if $u(i) \leq v(i)$, for each $i \in I_{\Sigma}$ (prefix ordering)

Recall that $u$ denotes a component vector while $u(i)$ denotes the sequence of the $i$-th coordinate of $u$ (and corresponds to the sequence of events on interface $i$ of the component).

Under these operations the set of component vectors $V_{\Sigma}$ is a monoid$^1$ with binary operation $\cdot$ and identity $A_{\Sigma}$, where $A_{\Sigma}$ is the unique vector with $A_{\Sigma}(i) = A$, for each $i \in I_\Sigma$. In other words, the component vector $A_{\Sigma}$ assigns the empty sequence to each interface of a component with signature $\Sigma$. The set of all possible component vectors $V_{\Sigma}$ is also a partially ordered set (poset) with ordering relation $\leq$ and bottom element $A_{\Sigma}$. These claims can be formally established with the following proposition. The proof picks up on the technique used in [Shi97] for behaviour vectors and is done by arguing coordinate-wise.

**Proposition 4.1.1.** The set of all possible vectors $V_{\Sigma}$ formed over a signature $\Sigma$ is

1. a monoid with binary operation $\cdot$ and identity $A_{\Sigma}$.

2. a poset with partial order $\leq$ and bottom element $A_{\Sigma}$.

$^1$Recall that a monoid is a semigroup with identity.
Chapter 4. Properties of a Component Language

Proof.

For (1), it suffices to show that \( V_\Sigma \) is closed under '\( \cdot \)' and that '\( \cdot \)' is associative. We argue coordinate-wise. Let \( u, v \in V_\Sigma \) and \( i \in I_\Sigma \). Since \( u(i), v(i) \in \beta(i)^* \) we have \( (u, v)(i) \in \beta(i)^* \). Hence, \( u, v \in V_\Sigma \), proving that \( V_\Sigma \) is closed under '\( \cdot \)'. Now, for associativity, if \( u, v, w \in V_\Sigma \), then for each \( i \in I_\Sigma \) we have

\[
(u, (v, w))(i) = u(i) (v, w)(i) = (u, v)(i) w(i) = (u, v)(i) w(i) = ((u, v), w)(i)
\]

Since \( (u, (v, w))(i) = ((u, v), w)(i) \), for all \( i \in I_\Sigma \), we have that \( u, (v, w) = (u, v), w \), so '\( \cdot \)' is associative.

For (2), we need to show reflexivity, antisymmetry and transitivity of '\( \leq \)'. Again, we argue coordinate-wise. Let \( u, v, w \in V_\Sigma \). Since \( u(i) \leq v(i) \), for all \( i \in I_\Sigma \), we have \( u \leq v \), giving reflexivity. If \( u \leq v \) and \( v \leq u \), then \( u(i) \leq v(i) \), for all \( i \in I_\Sigma \), and \( v(i) \leq u(i) \), for all \( i \in I_\Sigma \), so we deduce that \( u(i) = v(i) \), for all \( i \in I_\Sigma \), which implies that \( u = v \), proving antisymmetry. Finally, if \( u \leq v \) and \( v \leq w \), then \( u(i) \leq v(i) \), for all \( i \in I_\Sigma \), and \( v(i) \leq w(i) \), for all \( i \in I_\Sigma \), so \( u(i) \leq w(i) \), for all \( i \in I_\Sigma \), which in turn implies that \( u \leq w \), proving transitivity. \( \square \)

We note that a component language \( V \subseteq V_\Sigma \) is not a monoid in general as it is not closed under '\( \cdot \)' unless it contains the empty vector \( \Delta_\Sigma \). We will see in Section 4.2 that this is the case for discrete component languages.

In Chapter 3 we described a formal specification technique that uses UML for constraining the behaviour of a component. In particular, we used UML2.0 sequence diagrams to restrict to an appropriate subset \( V \) of \( V_\Sigma \), its component language, which captures constraints on the order in which events should occur on its interfaces. It was pointed out that a component is normally expected to engage more than once in the interaction given by a sequence diagram. To address this issue we introduced the notion of maximal component vectors and talked about a component language as a pattern of behaviour for a component.

Having defined prefix ordering and concatenation on component vectors, we may now characterise maximal component vectors in a component language.
4.1. Order Theoretic Properties

Definition 4.1.2. Let $V$ be a component language. A component vector $v \in V$ is a maximal component vector in $V$ if there is no other vector $u \in V$ such that $v \leq u$.

We shall write $\text{maxml}(V)$ for the set of all maximal component vectors in $V$. Put formally,

$$\text{maxml}(V) = \{ v \in V | \exists u \in V (u \neq v) : v \leq u \}$$

This definition reflects the fact that there are component vectors in $V$ which do not represent earlier behaviour than any other vector in $V$. Considering that a component language is a partially ordered set, the maximal component vectors are the upper bounds in $(V, \leq)$ [DP90].

With regard to the issue of repeating the interaction described in a sequence diagram, maximal component vectors serve to identify the point in which repetition may occur. In this sense, each coordinate of a maximal component vector can be seen as a recursive sequential CSP process description such as that used to specify controlled components in [ST04]. Another way to understand maximal component vectors is to view their coordinates as regular expressions of the form $\langle i \rangle^*$, where $v \in \text{maxml}(V)$ and '$\ast$' denotes the repetition operator in regular languages, as used for example in the behaviour protocols of [PV02].

It transpires that the $k$-th repetition of the behaviour prescribed in a sequence diagram results in component vectors whose coordinates comprise sequences prefixed by the sequences $\langle i \rangle$, for each $i \in I$, of the maximal component vectors, repeated $(k - 1)$ times. This reinforces the idea that analysis and reasoning about the observable behaviour of a component can be sufficiently performed at the level of the corresponding component language, i.e. within a finite set of behaviours.

Now based on the order theoretic properties of component vectors in $V \subseteq V_D$, we may introduce two further operations on a component language. These operations will be used in defining one of the properties, discreteness, that characterise well-behaved components. This will be discussed in Section 4.2. In addition, they are central to modelling concurrency between events on component interfaces and this will be described in Section 4.3.
Definition 4.1.3. Let $u$ and $v$ be component vectors in $V \subseteq V_{\Sigma}$, then

1. $u \sqcap v$ is defined to be the vector $w$ which satisfies $w(i) = \min(u(i), v(i))$, for each $i \in I_{\Sigma}$

2. $u \sqcup v$ is defined (if $\max(u(i), v(i))$ exists, each $i$) to be the vector $w$ which satisfies $w(i) = \max(u(i), v(i))$, for each $i \in I_{\Sigma}$

where $w(i), u(i), v(i)$ denote the $i$-th coordinate of vector $w, u, v$, respectively.

Note that these operations are computed coordinate-wise within our mathematical framework. For example, consider two component vectors $u, v$ where $u = (A, b_1 b_2, A)$ and $v = (a_1, b_1, A)$. To compute their least upper bound we apply (2) of Definition 4.1.3 and we get

$$u \sqcup v = (A, b_1 b_2, A) \sqcup (a_1, b_1, A) = (a_1, b_1 b_2, A) = u$$

Indeed, for the first coordinate (for $i = 1$) we have $u(1) = A, v(1) = a_1$, hence $w(1) = a_1$.

For the second coordinate (for $i = 2$), we have $u(2) = b_1 b_2, v(2) = b_1$, hence $w(2) = b_1 b_2$.

For the third coordinate (for $i = 3$), $u(3) = A, v(3) = A$, hence $w(3) = A$.

Thus, $w = (a_1, b_1 b_2, A)$. To compute the greatest lower bound of $u$ and $v$ we apply (1) of Definition 4.1.3 and we get

$$u \sqcap v = (A, b_1 b_2, A) \sqcap (a_1, b_1, A) = (a_1, b_1, A) = v$$

in a similar fashion.

In addition, and based on the observation that if $x, y, z$ are sequences such that $x, y \leq z$ then either $x \leq y$ or $y \leq x$, we may infer for component vectors that if $u, v \leq w$, each $i$, then $u(i), v(i) \leq w(i)$ so $u(i) \leq u(i)$ or $v(i) \leq v(i)$, each $i$. Now we may observe that if $u, v, w \in V$ such that $u, v \leq w$, then $u \sqcap v$ and $u \sqcup v$ are both defined. We shall use this fact in the sequel without further comment.

In terms of partial orders, the operations of Definition 4.1.3 essentially give the greatest lower bound and the least upper bound of $u, v \in V \subseteq V_{\Sigma}$, in the usual sense of lattices and domain theory [DP90]. Recall that if $(X, \leq)$ is a partially ordered set then the least
upper bound of \(x_1, x_2 \in X\), if it exists, is the least element \(x \in X\) such that \(x_1, x_2 \leq x\). We denote it by \(x_1 \sqcup x_2\). Dually, the greatest lower bound, denoted by \(x_1 \sqcap x_2\), is the largest element \(x \in X\) such that \(x \leq x_1, x_2\).

So, \(u \sqcup v\) of Definition 4.1.3 (2) is the least upper bound of \(u, v\) in \(V \subseteq V_\Sigma\) and \(u \sqcap v\) is the greatest lower bound of \(u, v\) in \(V \subseteq V_\Sigma\). Indeed, arguing coordinatewise, we have

\[
\exists \iff \forall i \in I_\Sigma, \exists(i) \leq u(i), v(i) \iff \forall i \in I_\Sigma, \exists(i) \leq \min(u(i), v(i)) \iff \forall i \in I_\Sigma, \exists(i) \leq (u \sqcap v)(i)
\]

Hence, \(\exists \leq u \sqcap v\).

Similarly, we may show that \(u \sqcup v\) is the least upper bound of \(u, v\) in \(V_\Sigma\) (replace \(\leq\) by \(\geq\), and \(\sqcup\) by \(\sqcap\), and \(\min\) by \(\max\)).

We next consider a right-cancellation operator \(\backslash\). Intuitively, if a component vector \(u\) describes behaviour which is an initial part of the behaviour described by another component vector \(v\), so that \(u \leq v\), then \(v/u\) is the 'continuation' of \(u\) that extends it to \(v\). This is formally put in the following definition.

**Definition 4.1.4.** Let \(u, v\) be component vectors in \(V \subseteq V_\Sigma\) with \(u \leq v\), then we define \(v/u\) to be the unique component vector \(\exists\) such that \(u \cdot \exists = v\).

The following result describes the basic interaction between the operations of Definition 4.1.3 and the right-cancellation operator.

**Proposition 4.1.2.** If \(u, v, w \in V_\Sigma\) such that \(u, v \leq w\), then \((u \sqcup v)/v\) and \(u/(u \sqcap v)\) both exist and are equal.

**Proof.**

We know that \(u \sqcup v\) is defined and we have \(v \leq u \sqcup v\) by definition of \(\sqcup\). Hence, \((u \sqcup v)/v\) is defined. Similarly, we know that \(u \sqcap v\) is defined and we have that \(u \sqcap v \leq u\) by definition of \(\sqcap\). Hence, \(u/(u \sqcap v)\) is defined.

Now we shall prove that they are also equal. Let \(i \in I_\Sigma\). Since \(u, v \leq w\), we have that \(u(i) \leq v(i)\) or \(v(i) \leq u(i)\).
If \( u(i) \leq v(i) \), then \( (u/(u \cap v))(i) = (u/v)(i) = A = (v/v)(i) = ((u \cup u)/v)(i) \).

If \( u(i) \leq v(i) \), then \( (u/(u \cap v))(i) = (u/v)(i) = ((u \cup v)/v)(i) \).

Thus, in either case we have that \( (u/(u \cap v))(i) = ((u \cup v)/v)(i) \), each \( i \), which implies that \( (u/(u \cap v)) = ((u \cup v)/v) \). □

Note that the hypothesis of the proposition is symmetric on \( u \) and \( v \) and thus, we would also have \( (u \cup u)/u = u/(u \cap u) \). The proof of this result first appeared in [SM04a] and we include it here as this proposition will be useful in establishing the relation between the notion of independence (cf Definition 4.3.1) and order theoretic properties such as the 'd' relation (cf Definition 4.2.5). This will be further discussed in Chapter 7 when we consider a class of automata in which concurrency is addressed as an explicit structural property.

4.2 Well-behaved Components

In this section, we use the mathematical properties of component vectors to identify a particular class of components, the so-called well-behaved components. Essentially, these are components whose language has the normality property. As we shall see in what follows, there are two aspects to normality; discreteness and local left-closure. Well-behavedness of the corresponding component reflects the fact that certain guarantees that accrue from discreteness and local left-closure of its component language are 'embedded' in its behaviour.

Apart from the theoretical motivation, discreteness and local left-closure can have practical benefits with regard to component-based design, as shown in [MSKF03, MKS05]. The idea is that in checking a component language against these conditions it is possible to identify missing behaviours - either undesirable or, simply, unthought.

The behaviour of a component captures assumptions about the order in which its operations can be called on its provided interfaces, and the order in which the component calls external operations through its required interfaces. Some services of a component may only be callable in certain situations. For instance, an initialisation service must
be called before other services become available. When a component is to be placed in a particular context, it is checked that the context satisfies these assumptions. On this evidence, the component is then expected to behave in predictable ways. Consequently, missing behaviours in the specification of a component may cause certain inconsistencies and as a result the system may exhibit pathological behaviour.

We illustrate how discreteness and local left-closure can be used to identify missing behaviours through an example case study in Section 4.4. These properties build on the order theoretic properties of component vectors presented in Section 4.1.

A key property of the sets of component vectors $V_\Sigma$ is that they possibly contain discrete subsets. Before defining discreteness, we also need to define consistent completeness.

**Definition 4.2.1.** We shall say that $V \subseteq V_\Sigma$ is consistently complete if and only if

- $\Delta_\Sigma \in V$
- whenever $u, v, w \in V$ and $u, v \leq w$, then $u \cup v \in V$.

In short, the notion of consistent completeness for a poset says that whenever two of its elements are less or equal than a third in the set, their least upper bound exists and is in the poset. Notice we have already seen that if $u, v, w \in V$ such that $u(i), v(i) \leq w(i)$, each $i$, then $u(i) \leq v(i)$ or $v(i) \leq u(i)$, each $i$. Hence, $\max(u(i), v(i))$ exists, each $i$, which means that we already have that $u \cup v$ exists. The point of Definition 4.2.1 is that, in addition, $u \cup v$ belongs to $V$. Now we may impose the first constraint on component languages.

**Definition 4.2.2.** Let $c = (\Sigma, V)$ be a component, then $V$ will be said to be discrete if and only if, $\Delta_\Sigma \in V$ and whenever $u, v, w \in V$ such that $u, v \leq w$ then

- $u \cup v \in V$
- $u \cap v \in V$

If the component language $V$ is discrete, then we shall say that component $c$ is discrete.
Note that the statement \( u \cup v \in V \) is understood as asserting that \( u \cup v \) is defined.

The definition of discreteness incorporates the notion of consistent completeness. The definition refers to component vectors in the language \( V \) of a component which have at least two distinct predecessors and says that both the least upper bound and the greatest lower bound of these predecessors must exist and also belong to its component language \( V \). In short, such vectors together with their predecessors must constitute finite lattices.

The justification for this constraint is as follows. The notion of discreteness derives from consideration of the construction described in Chapter 6. This construction allows us to translate a component language into an object called behavioural presentation [Shi88]. Behavioural presentations are order theoretic structures which generalise the event structures model [NPW81] in allowing time ordering of events to be a pre-order rather than a partial order, thereby allowing the representation of simultaneity as well as concurrency. Previous studies [Shi88, Shi97] uncovered a property which appeared to characterise the behaviour of discrete systems. This property determines a subclass of behavioural presentations, namely those that are discrete (cf Definition 6.2.5). Discrete behavioural presentations are described in Chapter 6, where we also show that discrete component languages can be associated with discrete behavioural presentations.

In fact, by requiring the discreteness property of a component language we constrain component behaviour in such a way that it can be guaranteed that: i) there are no infinite ascending or descending chains of occurrences of events, with respect to time ordering, which would give rise to Zeno type paradoxes, ii) there are no 'gaps' in the time continuum and iii) there is an initial point where nothing has happened. Exactly how i), ii) and iii) relate to the notion of discreteness shall become more clear when we discuss a behavioural presentation for a component in Chapter 6.

We also want to ensure that the behavioural presentation for a component contains one occurrence for each call to an operation on one of its interfaces. This can be guaranteed by a property called local left-closure, which we may justify as follows. Suppose that \( u \) is a component vector describing some behaviour. If \( i \in I_\Sigma \), then \( u(i) \) is the sequence of operation calls made at or from interface \( i \) during the course of this behaviour. If
there is a sequence \( x \) such that \( x < u(i) \), then there must be an 'earlier part' of \( u \), in which precisely the sequence \( x \) has taken place, at interface \( i \).

**Definition 4.2.3.** Suppose that \( c = (\Sigma, V) \) is a component. We shall say that \( V \) is locally left-closed if and only if, whenever \( u \in V \) and \( i \in I_D \) and \( x \in \beta(i)^+ \) such that \( A < x < u(i) \), then there exists \( y \in V \) such that \( y \leq u \) and \( v(i) = x \).

**Definition 4.2.4.** Suppose that \( c = (\Sigma, V) \) is a component. Its component language \( V \) will be said to be normal if and only if it is locally left-closed and discrete.

We also say that \( c \) is well-behaved, if \( V \) is normal.

Less formally, local left-closure says that whenever there exists a sequence of calls to operations (at an interface \( i \)) which is strictly less than that contained in a component vector at that interface, this sequence must be present at a coordinate of some other component vector which precedes the aforementioned one. The name of this property, local left-closure, comes from the very fact that locally, at the \( i \)-th coordinate of \( u \), the sequence \( u(i) \) is a prefix of \( u(i) \) that also belongs to the component language \( V \).

To anticipate, local left-closure guarantees that component vectors in \( V \) decompose into products of vectors, each of which has at most one operation call (event) per coordinate. These are the *column vectors* mentioned in Section 3.1.2 and will be formally defined below (cf Definition 4.2.6). It may be worth noting that column vectors correspond to (possibly simultaneous) event occurrences in the corresponding behavioural presentation and become particularly important when we attempt to establish a relation between the vector languages of components and automata. This is the main theme of Chapter 7 and we return to this discussion there.

Effectively, the local left-closure property is intended to resolve ambiguities that may arise from not having enough points to describe the course of the behaviour in question; not the start or the end, but the 'gaps' in between. In order to provide a precise description of a discrete behaviour we require that every occurrence of an event is 'recorded' in the vector language of the component. This implies the presence of a distinct *prime* element in \( V \) for each simultaneity class of occurrences, and for each appropriate interface. We shall be concerned with primes in Chapter 6 as they play
Chapter 4. Properties of a Component Language

a central role in associating component languages with behavioural presentations. For
the moment it suffices to understand that, in this context, the notion of prime refers to
component vectors which have a unique other vector immediately beneath them. Such
ordering among vectors in the component language \( V \) of a component is based on the
relation '\( \rho \)' which we define next.

**Definition 4.2.5.** Suppose that \( c = (\Sigma, V) \) is a component and \( u, v \in V \). We shall say
that \( u \) covers \( v \) in \( V \), and we write \( u <_V v \) if

1. \( u < v \) and \( u \neq v \)

2. If \( z \in V \) such that \( u < z \leq v \), then \( z = u \lor z = v \)

Subscript \( V \)'s will be omitted when the language is clear from context.

Intuitively, the relation '\( \rho \)' provides an ordering among elements of \( V \), in which one
is 'immediately beneath' the other, allowing no other vector in \( V \) to exist in between
them.

The following lemma relates the '\( \rho \)' relation to greatest lower and least upper bounds in
\( V \). This result will prove useful in identifying the complete primes in the poset \((V, \leq)\),
when it comes to associating a component language with a behavioural presentation in
Chapter 6. The need for this result was identified in [SM04b] but was not proven. We
include the proof here.

**Lemma 4.2.1.** Suppose that \( V \) is normal and that \( u, u_1, u_2, v, v_1, v_2 \in V \), then

1. If \( u < u_k \), each \( k \), and \( u_1 \neq u_2 \), then \( v_1 \cap v_2 = u \)

2. If \( u_k < u \), each \( k \), and \( u_1 \neq u_2 \), then \( v_1 \cup v_2 = u \)

**Proof.**

For (1), let \( w = v_1 \cap v_2 \). Since \( V \) is discrete, \( w \in V \) and \( u \leq w \leq u_k \), each \( k \). If \( w < u w \),
then as \( u < u_k \), each \( k \), we would have \( v_1 = w = v_2 \), a contradiction. Hence, \( w = u \).

For (2), let \( w = v_1 \cup v_2 \). Since \( V \) is discrete, \( w \in V \) and \( u_k \leq w \leq u \), each \( k \). If \( w < v_\),
then as \( u_k < u \), each \( k \), we would have \( u_1 = w = u_2 \), a contradiction. Hence, \( w = v \).
4.2. Well-behaved Components

which completes the proof. □

We now examine the structure of behaviours in a well-behaved component. The normality property imposes that the component language proceeds in discrete steps during which any event occurrences on its interfaces are captured in its component language. It turns out that each of the steps is such that, during the step, each interface has experienced at most one event (e.g. operation call). This motivates the following definition.

Definition 4.2.6. Suppose that $c = (\Sigma, V)$ is a component, then we define

$$E_\Sigma = \{e \in V_\Sigma \setminus \{A\} : i \in I_\Sigma \Rightarrow |e(i)| \leq 1\}$$

where for any sequence $x$, $|x|$ denotes the length of $x$.

We also define $E_\Sigma^1 = E_\Sigma \cup \{A\}$.

We shall refer to elements of $E_\Sigma$ as column vectors.

Intuitively, column vectors represent events occurring on component interfaces. They are a specific kind of component vectors in that each of their coordinates is either the empty sequence or a single event. In fact, $e \in E_\Sigma$ represents a simultaneity class of events. A simultaneity class of events is to be understood as a set of simultaneously occurring events. If there is a unique $i$ such that $e(i) > A$ (more simply, if $e$ has a unique non-empty coordinate), then the column vector represents a single event. Otherwise, it represents a simultaneity class of events; precisely those events $e(i)$ with $e(i) > A$. Such vectors correspond to simultaneity classes of event occurrences in the corresponding behavioural presentation. We return to this discussion when we describe a construction that maps component languages into behavioural presentations in Chapter 6. The following lemma asserts that behaviours in a normal (i.e. discrete and locally left-closed) component language proceed in such simultaneity classes.

Lemma 4.2.2. Suppose that $c = (\Sigma, V)$ is a component, $V$ is normal and let $u \in V$ with $u \neq A_\Sigma$, then there exists $v \in V$ and $e \in E_\Sigma$ such that $u = v \cdot e$ and $v \cdot e = u$.

Proof.
Since, $A_w < u$, the set $Y = \{y \in V : A \leq y < u\}$ is non-empty (it contains $A_w$). It is also finite, and hence contains a maximal element $y$, which is such that $y < u$. Thus, we have shown that there exists $y \in V$ such that $y < u$. We still need to show $u = y.e$.

Since $y < u$, we have $y(i) < u(i)$, some $i \in I_S$. We shall assume that $|u(i)| - |y(i)| > 1$ and reach a contradiction. Let $z \in \beta(i)^* \setminus \{A\}$ and $c \in \beta(i)$ such that $y(i) = z(i).c.e$. By local left-closure of $V$, there exists $w \in V$ such that $w \leq u$ and $w(i) = y(i).z$.

Since $w, y \leq u$, we have that $w \cup y$ is defined. Let $w' = w \cup y$. Since $V$ is discrete, we have $w' \in V$, and also $y \leq w'$. But we now have $y(i) < w(i).z = w'(i)$, giving $y < w'$. Also, we now have $w'(i) = w(i) \leq y(i).z < y(i).z.e = y(i)$, giving $w' < u$. Thus, $y < w' < u$, giving the required contradiction (since $y < u$). □

It might be instructive at this point to refer back to Section 3.1.2 where we first talked about component vectors as being built up by coordinatewise concatenation and revisit Example 3.1.3 in connection to this lemma.

The following result relates the '<' relation with the right-cancellation operator.

**Proposition 4.2.1.** Suppose that $c = (\Sigma, V)$ is a component, $V$ is normal and let $u, v \in V$. If $u < v$, then $v/u \in E_S$.

**Proof.**

Since $u < v$, we have $v/u \neq A_w$. We want to show that $v/u \in E_S$. If $v/u \notin E_S$, then for some $i \in I_S$, $u(i) = v(i).w_1.w_2$, where $w_1, w_2 > A$. By local left-closure, (and take $u(i).w_1$ as $z$) there exists $w \in V$ such that $w \leq u$ and $w(i) = y(i).w_1$.

Let $z = u \cup w$. Since $V$ is discrete we have $z \in V$, and also $u \leq z$. But now $u(i) < v(i).w_1 = w(i) = z(i)$ and hence, $u \leq z$. Also, $z(i) = w(i) = y(i).w_1 < u(i).w_1.w_2 = y(i)$ and hence, $z \leq v$. This implies that $u < z < v$, which is a contradiction (since $u < v$). □

To anticipate further, these results allow us to define a transition structure on a component language $V$ when it comes to associating components with automata in Chapter 7.
4.3 Relations between Events in a Component Language

So far, we have seen that a component vector provides a snapshot of behaviour in which the component has experienced the events appearing on the vector's coordinates. Component vectors can be seen to be built up from the empty vector by a series of concatenations with column vectors which represent events. In a component language we only keep those vectors corresponding to events that we expect the component to engage in during the course of its intended behaviour. Such events were identified using UML sequence diagrams in Chapter 3 (Section 3.2) and we saw that they may occur sequentially, concurrently, simultaneously or may be mutually exclusive.

In this section, we describe how the various relationships between events occurring on component interfaces manifest themselves in the order structure of the corresponding component language. It transpires that the relationships between events occurring on different interfaces of components are determined based on what other component vectors appear in the language. We show that by exploiting the basic order theoretic properties of component vectors we may talk about events occurring in sequence, concurrently, simultaneously, or being in conflict.

4.3.1 Events in sequence

The prefix ordering among component vectors can be viewed as an ordering on partial executions, where each component vector corresponds to that portion of behaviour in which the component has already engaged in the events appearing on its coordinates. This can be expressed more succinctly by saying that \( u \preceq v \) in a component language means that \( u \) is an earlier part of behaviour leading to \( v \).

If in addition the component language is normal, then we can say more than that. In particular, we have seen (Proposition 4.2.1) that whenever \( u \) covers \( v \) in a normal component language, then what takes \( u \) and 'stretches it up' to \( v \) is a column vector representing the occurrence of an event (or, more precisely, a simultaneity class of events). This allows us to model causally related events. That is, occurrence of one event depends on the previous occurrence of the other. It is in this sense that we talk
about events occurring in sequence (one after the other).

Suppose that the component has experienced a series of events and the resulting behaviour is described by a component vector $\mathbf{u} = (a_1, A, c_1, A)$. Then, occurrence of $\mathbf{e}_1 = (A, b_1, A, A)$ followed by occurrence of $\mathbf{e}_2 = (a_2, A, A, d_1)$ can be modelled by

- first, concatenating vector $\mathbf{u}$ with $\mathbf{e}_1$, and
- then, concatenating the resulting vector $\mathbf{y}$ with $\mathbf{e}_2$.

In terms of our mathematical framework this amounts to operations $\mathbf{u} \cdot \mathbf{e}_1 = \mathbf{v}$ and then $\mathbf{v} \cdot \mathbf{e}_2 = \mathbf{w}$.

Considering the Hasse diagram for the order structure of the corresponding component language $V$, where lines between vectors denote an ordering relation in which the topmost vector is greater than the lower one, this would result in the portion of the diagram shown in Figure 4.1.

```
(1, 2, 3, 4)
```

Figure 4.1: Event $b_1$ followed by simultaneous events $a_2$, $d_1$

It is important to make the observation that the actual ordering between events appearing in different coordinates of a component vector is determined by context - by what other vectors are included in the language. In other words, the relationship between component vectors and associated order theoretic structures is very much dependent
on what other vectors are in the set $V$ (unlike the behaviour vectors in [Shi79, Shi97] where this relationship is independent of context).

For instance, in component vector $y = (a_1, b_1, c_1, A)$ we may immediately derive that event $a_1$ has happened on the interface corresponding to the first coordinate, event $b_1$ has happened on the interface corresponding to the second coordinate and event $c_1$ has happened on the interface corresponding to the third. To determine the relationship between these events however, we need the rest of the language.

Assume that $V$ is given by the set

$$V = \{(A, A, A, A), (a_1, A, c_1, A), (a_1, b_1, c_1, A), (a_1a_2, b_1, c_1, d_1)\}$$

Notice that adding in $(A, A, A, A)$ is essential and in this case suffices for making $V$ discrete and locally left-closed. Now the presence of $y = (a_1, A, c_1, A)$ for which $y < v$, tells us that event $b_1$ on the interface corresponding to the second coordinate occurs only after both $a_1$ and $c_1$ have taken place. Further, the presence of $A_{\Sigma_{ex}} = (A, A, A, A)$ for which $A_{\Sigma_{ex}} < y$ dictates that $a_1$ on the interface corresponding to the first coordinate and $c_1$ on the interface corresponding to the third coordinate occurred simultaneously.

Now suppose that the intended behaviour of the component prescribed that $a_1$ must occur before $c_1$. This is captured in the corresponding component language by adding in the vector $(a_1, A, A, A)$. In the resulting language

$$V = \{(A, A, A, A), (a_1, A, A, A), (a_1, A, c_1, A), (a_1, b_1, c_1, A), (a_1a_2, b_1, c_1, d_1)\}$$

which continues to be normal. The presence of $w = (a_1, A, A, A)$ for which $w < u$ dictates that $a_1$ on the interface corresponding to the first coordinate occurs strictly before $c_1$ does on the interface corresponding to the third coordinate.

### 4.3.2 Concurrent events

The way component vectors are formed (as tuples of sequences) bears some relevance to the interleaving operator `||` of CSP [Ho95]. The semantics of 'process $P_1$ interleave process $P_2$' in CSP, denoted by $P_1||P_2$, says that concurrent execution of $P_1$ and $P_2$ requires no synchronisation. In the case of common events (i.e. both processes are able
to perform the same event), only one of the processes will engage in any particular occurrence of a common event.

Distinct coordinates in a component vector seem to adhere to something like the '|||' operator in the sense that they execute independently of each other. The difference is that on concurrent events (and if these can be understood as the equivalent of common events between processes in CSP) we do not assert that only one of the coordinates engages in it. Each corresponding interface experiences its share of the concurrent events, i.e. the event in its alphabet $\beta(i)$. Hence, the component as a whole experiences both events, and the ordering is irrelevant. This reflects the fact that concurrent events are in no way constrained to happen in any particular order, including simultaneity. In the resulting behaviour, after both events have taken place, we say that the two events are concurrent.

Our approach draws upon the concept of Mazurkiewicz traces [Maz77] where the ordering of concurrent events is considered subjective and thus is not distinguished, in contrast to CSP trace theory where it is assumed that observations are sequential in nature leading to the interpretation that concurrent events occur in either order.

For systems that exhibit concurrency, different external observers may disagree on the ordering of concurrent events. This may be seen more clearly in Einstein's famous thought-experiment\(^2\) involving two trains travelling at constant speed in opposite directions along a pair of parallel tracks. Observers $O_1$ and $O_2$ are sitting in the middle of each train. A third observer $O_3$ is sitting on the embankment. At a given moment, the two observers on the trains are on a line at right angles to the third observer. At that moment, two bolts of lightning strike on either end of the first train in such a way that $O_3$ sees them strike at exactly the same time. Observer $O_1$ travelling towards the light coming from the strike on the front end of the train he is on, sees that light before he sees the light of the strike on the rear end of the train. Observer $O_2$ travelling towards the light coming from the strike on the rear end, sees that light before she sees the light coming from the strike on the front end.

\(^2\)This thought-experiment was given by A. Einstein in [Ein21] to demonstrate the non-objectivity of contemporaneity in relativistic mechanics. It has been considered in view of concurrency in [Shi97] and our description of the experiment here is based on that.
Now from the point of view of observer $O_1$ there are three distinct behaviours of the "system". One is when nothing has happened yet, another when he has seen the lightning bolt from the front end of the train, and another when he has seen both lightning bolts. Likewise, observer $O_2$ has seen a behaviour in which nothing has happened yet, a behaviour where she has seen the lightning strike on the rear end and a behaviour where she has seen both lightning bolts. From the point of view of observer $O_3$ there are only two distinct behaviours. One is when nothing has happened yet and the other is when both have. Thus, all four distinct behaviours can be observed for the same system; nothing has happened, one event has happened, the other event has happened, and both events have happened.

The point to be made here is that observations on systems exhibiting concurrency largely depend on the relative position of the observer or the actual timing of execution. Such differences are non-objective and do not allow to infer the actual ordering between the events. On this basis, any particular ordering between concurrent events is irrelevant. On the contrary, the ordering between causally related events is objective (independent of the observer) and should be distinguished.

Returning to the treatment of concurrency within our component model, this takes up on Mazurkiewicz trace languages [Maz77, Maz88], which introduce additional structure into formal languages in order to describe non-sequential behaviour. The additional structure is given in terms of an independence relation, over action symbols (understood as events here), which describes potential concurrency.

Let $A$ denote a (finite) set. A concurrent alphabet is an ordered pair $(A, \iota)$ where $\iota \subseteq A \times A$ satisfies:

- $a \iota b \Rightarrow b \iota a$ (symmetry)
- $a \iota b \Rightarrow a \neq b$ (irreflexivity)

Symmetry requires that concurrency is always mutual while irreflexivity prohibits an event being concurrent with itself.

Component vectors are essentially tuples of sequences, as discussed before. Thus, we find it useful to consider the extension of the relation $\iota$ to sequences, based on [Maz77].
Given a concurrent alphabet \((A, \iota)\), a relation \(\equiv_{\iota}^{(1)}\) can be defined on \(A^*\) by

\[
x \equiv_{\iota}^{(1)} y \iff \exists u, v \in A^*, \exists a, b \in A \text{ such that } aub \land x = uabv \land y = ubav
\]

Let \(\equiv_\iota\) be the reflexive, transitive closure of \(\equiv_{\iota}^{(1)}\). By definition, \(\equiv_\iota\) is an equivalence relation on \(A^*\). We denote the equivalence class of \(x \in A^*\) by \(< x >\). The set of equivalence classes of \(A\) with independence relation \(\iota\) is denoted by \(A_\iota^* = \{< x > | x \in A^*\}\). Any subset \(L\) of \(A_\iota^*\) is a Mazurkiewicz trace language.

Thus, the independence relation \(\iota\) on \(A\) gives rise to an equivalence relation \(\equiv_\iota\) on sequences formed over \(A\). We make use of this construction in terms of sequences formed over the sets of events \(\beta(i)\), for each \(i \in \mathcal{I}_\Sigma\), associated with component interfaces.

Intuitively, the equivalence relation on sequences of events says that any two consecutive events are allowed to permute, providing they are independent. Note that when the independence relation is empty in the sets \(\beta(i)\), all \(i \in \mathcal{I}_\Sigma\), no two events can be permuted in the corresponding sequences \(\beta(i)^*\), which amounts to our understanding of sequential systems (e.g. as described by processes in CSP [Hoa85]).

The equivalence relation on \(Op_\Sigma^*\) equates all, and only those, sequences from \(\beta(i)^*\), for each \(i \in \mathcal{I}_\Sigma\), which differ in the order of concurrent events. Drawing upon the extension of the independence relation \(\iota\) to behaviour vectors in [Shi97], the notion of independence between events in Mazurkiewicz traces can be readily interpreted into component vectors in our approach.

**Definition 4.3.1.** For \(u, v \in V \subseteq V_\Sigma\), we define

\[
y \text{ ind } v \iff \forall i \in \mathcal{I}_\Sigma : u(\iota) > \iota \Rightarrow v(\iota) = \iota
\]

If \(u \text{ ind } v\), then we will say that \(u\) and \(v\) are independent.

The intuition is that the behaviours described by \(u\) and \(v\) may occur independently. In the case of column vectors, independence captures the fact that events appearing in one vector may occur independently of those appearing in the other. If in addition they occur consecutively, then they are concurrent. Thus, whenever two consecutive events permute, their corresponding column vectors commute, i.e. \(e_1 \cdot e_2 = e_2 \cdot e_1\), and the resulting behaviours are concurrent. In fact, \((E_\Sigma, \text{ ind})\) is a concurrent alphabet.
For example, suppose that a component with 3 interfaces has experienced a fragment of behaviour described by \( u = (a_1, A, A) \) and after that may engage in \( e_1 \) and \( e_2 \) concurrently, where \( e_1 = (A, d_1, A) \) and \( e_2 = (A, A, e_1) \). This is in fact the intended behaviour of component \( B \), as described in Figure 3.6 of Example 3.2.3 (in Section 3.2.5). We make the observation that \( e_1 \) and \( e_2 \) and consequently,

\[
e_1 \cdot e_2 = (A, d_1, A). (A, A, e_1) = (A, d_1, e_1) = (A, A, e_1). (A, d_1, A) = e_2 \cdot e_1
\]

Thus, we have \( u \cdot e_1 \cdot e_2 = w = u \cdot e_2 \cdot e_1 \). Indeed,

\[
u \cdot e_1 = (a_1, A, A). (A, d_1, A) = (a_1, d_1, A) = v_1
\]

and

\[
u_1 \cdot e_2 = (a_1, d_1, A). (A, A, e_1) = (a_1, d_1, e_1) = w
\]

We also have that

\[
u \cdot e_2 = (a_1, A, A). (A, A, e_1) = (a_1, A, e_1) = v_2
\]

and

\[
u_2 \cdot e_1 = (a_1, A, e_1). (A, d_1, A) = (a_1, d_1, e_1) = w
\]

In the resulting behaviour \( w \) the events \( d_1 \) and \( e_1 \) are concurrent. The situation is depicted in Figure 4.2.

```
\[
\begin{array}{c}
\vdots
\end{array}
\]

\[
\begin{array}{c}
(a_1, d_1, c_1) = w
\end{array}
\]

\[
\begin{array}{c}
\vdots
\end{array}
\]

\[
\begin{array}{c}
y_1 = (a_1, d_1, A) \quad (a_1, A, e_1) = y_2
\end{array}
\]

\[
\begin{array}{c}
(a_1, A, A) = y
\end{array}
\]

\[
\begin{array}{c}
\vdots
\end{array}
\]

Figure 4.2: Concurrent events \( d_1 \) and \( e_1 \)

Chapter 4. Properties of a Component Language

Note that if the events were not concurrent, then we would have the lower part of the diamond shape in the diagram but not the upper half. The upper half is obtained only when the column vectors corresponding to the events in question commute (that is to say, equivalently, that they are independent) and represent events that occur consecutively. Both these requirements have to be met for the events to be concurrent. This is then reflected in the order structure of the corresponding component language by the presence of the vector forming the upper half of the diamond. This vector is the resulting common behaviour, after the concurrent events have taken place. If either of these two requirements is violated, then the component would never exhibit the common behaviour described by \( w \). The point to be made here is that independence alone does not guarantee concurrency. (The case of non-independence is more obvious.)

As depicted in Figure 4.2, the component as a whole experiences both events (on each appropriate interface) and the ordering is irrelevant. The corresponding concatenations result in a unique component vector (sitting on the top of the diamond) in which both events have occurred in no particular order. The incomparable component vectors in the middle of the diamond (i.e. \( v_1, v_2 \)) represent behaviour arising during concurrent execution. These two vectors are bounded above by the vector in which both concurrent events appear (i.e. \( w \)). In terms of the order theoretic properties of component vectors discussed in Section 4.1, this vector is their least upper bound. In the example of Figure 4.2, we have

\[
v_1 \cup v_2 = (a_1, d_1, A) \cup (a_1, A, c_1) = (a_1, d_1, c_1) = w
\]

Their greatest lower bound (sitting at the bottom of the diamond) is the vector in which none of the concurrent events have occurred but are both available. In our example, we have

\[
v_1 \cap v_2 = (a_1, d_1, A) \cap (a_1, A, c_1) = (a_1, A, A) = \mathcal{Y}
\]

We shall see that this non-interleaving representation of concurrent behaviour manifests itself in the structure of the associated automata for components, described in Chapter 7.

The fundamental difference in expressing concurrency should now be apparent. By departing from classic CSP concurrency, we are able to consider concurrency within a
single component. In CSP, and related process algebras, concurrency arises through composition. Here we have not yet been concerned with composing sequences from different components' interfaces, though this may also produce concurrency. We are simply describing the case that interfaces of the same component engage in concurrent events, a phenomenon common in reactive systems for example. We shall see how component vectors are composed within our framework in Chapter 5. The notion of composition of component vectors from different components will then be discussed in view of the synchronisation parallel operator $||$ of CSP [Hoa85].

We will also see how concurrency in individual components is carried through to the composite component, as an explicit structural property of the associated automata, in Chapter 7.

In what follows, we again discuss concurrent events in connection to the context of the corresponding component language. Consider the component language

$$V = \{(A, A, A), (a_1, A, A), (a_1, A, 1), (a_1, 1, A), (a_1, 1, e_1), (a_1, d_1, e_1)\}$$

It can be easily checked that $V$ is discrete and locally left-closed. Its order structure is (in part) depicted in Figure 4.2. We have that $u \prec v_1$ and $u \prec v_2$. Also, $v_1 \prec w$ and $v_2 \prec w$. We have seen that the events $d_1$ on the second interface and $e_1$ on the third are concurrent.

Now consider the component language,

$$V = \{(A, A, A), (a_1, A, A), (a_1, d_1, e_1)\}$$

In this language, which is also discrete and locally left-closed, the events $d_1$ on the second interface and $e_1$ on the third are simultaneous rather than concurrent. This is because the component vector $w = (a_1, d_1, e_1)$ in which both events have taken place is obtained directly from $u = (a_1, A, A)$ in which neither of the events have occurred yet. Hence, what takes $u$ and stretches it up to $w$ is the column vector $g = (A, d_1, e_1)$ in which $d_1$ and $e_1$ are simultaneous event occurrences. This case can be understood as cutting through the diamond of Figure 4.2.

Next, consider the component language

$$V = \{(A, A, A), (a_1, A, A), (a_1, d_1, A), (a_1, A, e_1)\}$$
In this language, which is also discrete and locally left-closed, the events \( d1 \) and \( e1 \) are neither concurrent nor simultaneous. The component in this case, after doing \( a1 \) on the interface corresponding to the first coordinate, has a choice between doing \( d1 \) on the second coordinate or \( e1 \) on the third. This case can be understood as having only the lower half of the diamond in Figure 4.2, and brings about the issue of alternative events and mutual exclusion. This is discussed in the following section.

4.3.3 Events in conflict

Based on the prefix ordering between component vectors in the set \( V_2 \), we may also model events which are in conflict. That is, events which are mutually exclusive in that occurrence of one excludes occurrence of the other.

In discussing concurrent events, we saw that the two incomparable component vectors in the middle of the diamond represent concurrent behaviour. The fact that the two incomparable vectors are in the middle of the diamond implies that they are bounded above in the set (by the component vector sitting on top of the diamond).

Whenever this latter requirement does not hold we may talk about events in conflict. In terms of pictures and associated Hasse diagrams, we are essentially getting rid of the upper part of the diamond and keeping the lower part, the branches of which represent a choice between performing one or the other event. In effect, this amounts to ensuring that the behaviours they represent is not (an early) part of the same behaviour. Therefore, in what follows we examine when two component vectors are not bounded above in a component language.

Let's first consider the case where the column vectors in question are not independent. Then, they do not agree on the non-empty coordinates corresponding to the same interfaces. This entails that there is no causality between the two and at the same time it is not possible for both of them to occur (since they engage the same interfaces of the component).

For example, the events represented by \( g_1 = (a1, A, A) \) and \( g_2 = (a2, A, A) \) could both be available (or possible to occur) when the component has already engaged in the
4.3. Relationships between Events in a Component Language

behaviour described by a component vector, say, \( \mathbf{u} \). In other words, after \( \mathbf{u} \) the component may engage in either \( \mathbf{e}_1 \) or \( \mathbf{e}_2 \). Note that it cannot do both since \( a_1, a_2 \) \((a_1 \neq a_2)\) are events of the same interface (the one corresponding to the first coordinate). Considering the Hasse diagram for the order structure of the corresponding component language, this situation would result in the fragment of the diagram shown in Figure 4.3.

\[
\begin{align*}
\mathbf{y}_1 &= (a_1, b_1, A) \\
&
\mathbf{y}_2 &= (a_2, b_1, A) \\
\end{align*}
\]

\[
(A, b_1, A) = \mathbf{u}
\]

\[
\therefore 
\]

Figure 4.3: Events \( a_1 \) and \( a_2 \) are in conflict

In further explanation, \( \mathbf{y}_1 = (a_1, b_1, A) \) is the behaviour resulting from occurrence of \( \mathbf{e}_1 \) while \( \mathbf{y}_2 = (a_2, b_1, A) \) is the behaviour resulting from occurrence of \( \mathbf{e}_2 \), after \( \mathbf{u} \). In terms of our mathematical framework, we have \( \mathbf{u} \cdot \mathbf{e}_1 = \mathbf{v}_1 \) and \( \mathbf{u} \cdot \mathbf{e}_2 = \mathbf{v}_2 \) but only one of these behaviours may take place during an execution of the component in question. A client (e.g. another component) expecting \( \mathbf{y}_1 \) may be disappointed.

We now turn our attention to events whose corresponding column vectors are independent. This case is a bit more subtle. In principle, independent column vectors represent events which are in no way related to each other. For example, consider the events given by \( \mathbf{e}_1 = (a_1, A, A) \) and \( \mathbf{e}_2 = (A, A, c_1) \). If they are both offered after the component has engaged in behaviour described by \( \mathbf{u} \), then they represent a choice between doing \( a_1 \) on the interface corresponding to the first coordinate and event \( c_1 \) on the interface corresponding to the third coordinate. Unless they are bounded above!

To ensure that the two independent events are not bounded above, effectively, they are not part of a subsequent common behaviour, they must not occur consecutively. In other words, the event succeeding \( \mathbf{e}_1 \) must not be \( \mathbf{e}_2 \) and, dually, the event succeeding
\( \varepsilon_2 \) (on the other branch) must not be \( \varepsilon_1 \). Otherwise, they lead to a common behaviour \( \varepsilon \) which inadvertently bounds \( \varepsilon_1 \) and \( \varepsilon_2 \) (forcing them to be concurrent as discussed before).

The situation is depicted in Figure 4.4(i) where the events \( a_1 \) and \( c_1 \) are in conflict. Compare with Figure 4.4(ii) where the events \( a_1 \) and \( c_1 \) are concurrent.

\[
\begin{align*}
\varepsilon_1 &= (a_1, b_1, A) \\
(A, b_1, c_1) &= \varepsilon_2 \\
(A, b_1, A) &= \varepsilon \\
& \vdots \\
\end{align*}
\]

Figure 4.4: Events \( a_1 \) and \( c_1 \) are concurrent in (ii) but not in (i)

This should become more clear with the formal treatment of conflicting events within the context of the behavioural model for components presented in Chapter 6.

### 4.4 Illustration by example

We illustrate our approach to formally describing a component by means of an extended example inspired from component-based software used in consumer electronics products. The example comprises software components supporting a TV platform (see Figure 4.5) and is actually an extended version of the real-life example presented in [vO03]. We shall use it as a running example to illustrate our formal approach throughout the thesis.

We will be concerned with the MENU functionality of a TV set and in particular, the task of tuning the TV to a given frequency. Figure 4.5 depicts the component
4.4. Illustration by example

![Component specification architecture for a TV platform](image)

Figure 4.5: Component specification architecture for a TV platform

specification architecture using UML with regard to the tuning task. The stereotype `<component>` is used to describe component specifications and the UML 2.0 'ball and socket' notation is used for matching provides/requires interfaces from different components. The component architecture of Figure 4.5 comprises a set of application-level components together with their structural relationships and behavioural dependencies.

For the moment, we will be focusing on the MANUAL STORE option of the MENU functionality of a TV set. Figure 4.6 depicts a sketch of the screen displayed to the user with regard to the MANUAL STORE option of the TV. At the user level, the MANUAL STORE menu items are highlighted in turn by an 'arrow down' event which takes place through the TV remote control. Once an item is highlighted, its functionality is activated by an 'arrow right' or 'arrow left' event, again via the TV remote control.

The Search option allows the user to initiate a frequency search to detect the signal of a TV channel. Once a signal is found, the corresponding frequency value (e.g. 451) is displayed next to the Search option. The Fine Tune menu item provides the user with the option of adjusting the frequency value of a channel to optimise the signal.
Figure 4.6: The MANUAL STORE menu at the user level

reception. The Fine Tune value is initially set to 0 and the user can increment it by one via an 'arrow right' event or decrement it by one via an 'arrow left' event. These functionalities are made available to the user by the CMenu component.

The MANUAL STORE options are provided by the interaction of the CMenu and CTuner components of Figure 4.5. We isolate these components in Figure 4.7, where names for the interfaces are also included.

Figure 4.7: Component specification architecture

The CMenu component establishes communication with users via its provided interfaces ISearchFre and IFineTune. The ISearchFre interface has operations highlightSItem() and startSearch(). Calls to these operations shall be denoted by a1, a2 respectively, for abbreviation. The IFineTune interface has operations highlightFItem(), incrementFre() and decrementFre(), abbreviated by b1, b2 and b3 respectively. A user requests to search the available frequency for a program via the ISearchFre interface. The CMenu component cannot satisfy the requested operation itself and requires a component providing the IDetectSignal interface to conduct the frequency search on its behalf. This
is done by invocation of an operation detectSignal() (abbreviated by cl) on its required interface IDetectSignal, which is implemented by the CTuner component.

In what follows, we apply the formalism presented in Section 3.1 to model the CMenu component. Its signature is given by \( E_M = (P_M, R_M, \beta_M) \) where \( R_M = \{IDetectSignal\} \) is the set of required interfaces and \( P_M = \{ISearchFreq, IFineTune\} \) is the set of provided interfaces of CMenu. Consequently, the complete set of interfaces is given by the set \( I_{EM} = \{ISearchFreq, IFineTune, IDetectSignal\} \) and of course, \( P_M \cap R_M = \emptyset \).

The function \( \beta \) as defined in Definition 3.1.1 provides the set of operations associated with each interface. In this case we have,

\[
\begin{align*}
\beta_M(ISearchFreq) &= \{a1, a2\} \\
\beta_M(IFineTune) &= \{b1, b2, b3\} \\
\beta_M(IDetectSignal) &= \{cl\}
\end{align*}
\]

Suppose that a component developer considers the expected behaviour of CMenu fulfilling the following:

- The Fine Tune option should be highlighted before the user can change the default fine tune value.

- The Search option should be highlighted before the user can request a frequency search.

- Once the user requests a search (which corresponds to invoking operation \( a2 \) on interface ISearchFreq) the CMenu component requires a service from the CTuner component (calling operation \( cl \) on interface IDetectSignal).

- An occurrence of an operation call \( a2 \) on ISearchFreq should be followed immediately by an operation call \( cl \) on IDetectSignal, and nothing should be allowed to happen in between.

Such informal description of behaviour for a component is often accompanied by sequence diagrams that describe the series of interactions the component should perform for its correct participation in fulfilling the specific task.
Given the informal description of behaviour for the CMenu component, we have the UML sequence diagram of Figure 4.8, which describes the allowed sequence of events on its interfaces with respect to the task of tuning. Notice that the mode of communication is synchronous within a TV platform and this is represented by the use of filled arrowheads in the diagram.

![UML sequence diagram of Figure 4.8](image)

**Figure 4.8:** The CMenu component performing the tuning task

By unfolding the sequence diagram into component vectors, following the formal construction described in Section 3.2, we may obtain the set of behaviours that indicate the intended behaviour of the CMenu component, i.e. its component language. To avoid overelaborating the example, we do not give the details of the unfolding here - it can be easily checked against the material and examples of Section 3.2.

If we write \((x, y, z)\) for the function \(y\) of Definition 3.1.2 with \(y(\text{SearchFreq}) = x\),
4.4. Illustration by example

\[ y(IFineTune) = y \text{ and } y(IDetectSignal) = z \]\n
we obtain the following set of component vectors which comprise the component language of CMenu.

\[
V_M = \{(\Lambda, \Lambda, \Lambda), (a1, a1, a1), (a1, b1, \Lambda), (a1a2, a1, c1), (a1, b1b2, a), (a1, b1b3, a), (a1a2, b1, c1), (a1a2, b1b2, c1), (a1a2, b1b3, c1)\}
\]

Hence, the CMenu component is given by \( c_M = (\Sigma_M, V_M) \) (recall Definition 3.1.3) where \( \Sigma_M = (P_M, R_M, \beta_M) \) is the component signature and \( V_M \) is a subset of all component vectors, \( V_{\Sigma_M} \), formed over \( \Sigma_M \).

We now turn our attention to well-behavedness of the CMenu component. This entails considering the normality property in its component language. In what follows we examine discreteness and local left-closure of the CMenu component and illustrate why these properties are important in component design. The idea is that, from an initial set of component intended behaviours provided by the component designer(s), our proposed formal framework can determine whether this set is complete or on the contrary possible (and potentially faulty) scenarios have been omitted. In our framework, we can determine this by checking whether the component is well-behaved which corresponds to checking the properties of discreteness and local left-closure of the corresponding component language. Again, the advantages in doing so are that we may identify missing behaviours (either undesirable or simply unthought).

We start by considering the order structure of the elements in the set \( V_M \) of CMenu. This is depicted in the Hasse diagram of Figure 4.9.

It can be seen in Figure 4.9 that vectors \((a1a2, b1b2, c1), (a1a2, b1b3, c1), (a1, b1b2, \Lambda)\) and \((a1, b1b3, \Lambda)\) are the maximal component vectors, in the sense that they do not describe earlier behaviour than any other vector in \( V_M \). Likewise, vector \((\Lambda, \Lambda, \Lambda)\) is the minimal component vector representing behaviour of the component in which nothing has happened.

This implies that after reaching one of the maximal behaviours, say \((a1a2, b1b2, c1)\), the component can only continue by repeating the specific task from the beginning.
In such a case, its intended behaviour is again given by the vectors in its component language $V_M$, but this time instead of starting from $(A, A, A)$ each coordinate will be prefixed by the corresponding sequence appearing in the maximal component vector visited in the previous iteration. In this sense, the component language is providing a pattern of behaviour which the component must follow at all times.

Now based on Figure 4.9, we examine the discreteness property of the CMenu component. In order to do so, we concentrate on component vectors in $V_M$ with at least two distinct incomparable predecessors. They, together with their predecessors should constitute (finite) lattices, according to Definition 4.2.2 of discreteness. In other words, whenever two component vectors are less or equal than a third, also in $V$, their least upper bound must exist and belong to the set and their greatest lower bound must also be in the set.
4.4. Illustration by example

By careful examination of Figure 4.9 it can be seen that the discreteness property is violated in the component language of CMenu. In particular, vectors \((a_1, b_1, \Lambda)\) and \((a_1a_2, \Lambda, \Lambda)\) are less or equal than \((a_1a_2, b_1, c_1)\). This implies that:

- their greatest lower bound must be in the set; their greatest lower bound is \((a_1, b_1, \Lambda) \cap (a_1a_2, \Lambda, \Lambda) = (a_1, \Lambda, \Lambda)\).

- their least upper bound should belong to the set; their least upper bound is given by \((a_1, b_1, \Lambda) \cup (a_1a_2, \Lambda, \Lambda) = (a_1a_2, b_1, \Lambda)\).

The vector \((a_1, \Lambda, \Lambda)\) is in the set \(V_M\), but the vector \((a_1a_2, b_1, \Lambda)\) is not. Thus, according to our mathematical framework this vector should be added in order to make the component language \(V_M\), and consequently the CMenu component, discrete. By adding in vector \((a_1a_2, b_1, \Lambda)\) we get the following set \(V_M\). Its order structure is depicted in the diagram of Figure 4.10.

\[
V_M = \{(\Lambda, \Lambda, \Lambda), (a_1, \Lambda, \Lambda), (a_1, b_1, \Lambda), (a_1a_2, \Lambda, \Lambda), (a_1a_2, \Lambda, c_1), (a_1, b_1b_2, \Lambda), (a_1, b_1b_3, \Lambda), (a_1a_2, b_1, c_1), (a_1a_2, b_1b_2, c_1), (a_1a_2, b_1b_3, c_1), (a_1a_2, b_1, \Lambda)\}
\]

As can be seen in the Hasse diagram of Figure 4.10, the process of checking against discreteness has indicated potential concurrency between two pairs of events:

- \(a_2\) and \(b_1\)
- \(b_1\) and \(c_1\)

The first case amounts to the user pressing the 'arrow down' and the 'arrow right' buttons of the TV remote control together (concurrently). Within reason, this is something a user would not be expected to do. The CMenu component can in principle accept both events concurrently (since they engage distinct interfaces) but should not do so. To adhere to the behaviour prescribed by the component designer, it should deal with \(a_2\) first (including calling \(c_1\) on CTuner) and then deal with \(b_1\).
The second case is more subtle and does not involve any expectations of the user. It says that the user might request to fine tune the reception of a signal (and this is legitimate since it is requested after a2) at the same time that CMem is still managing its own dependencies on CTuner (requesting a frequency search via c1). This does not meet the product requirement that says that an event a2 must be followed by event c1 and nothing must be allowed to happen in between. Thus, the component vector (a1a2, b1, A) can be regarded as describing an instance of pathological behaviour.

Note that this requirement is not guaranteed in the behaviour arising from the interaction described in the sequence diagram of Figure 4.8 because there is no way to enforce that the user (represented by the human icon in the diagram) is aware that c1 has occurred before issuing b1. This is because the sending of c1 and the sending of b1 originate in different component lifelines and thus involve different components. This situation is often referred to as a race condition.
In any case, perhaps even more so in the second, our formal model is issuing a warning to the component designer(s). The diagram of Figure 4.10 says that in the course of exhibiting the intended behaviour, described by vectors \((a1a2, b1b2, c1)\) and \((a1a2, b1b3, c1)\), the CMenu component might exhibit the potentially undesirable behaviour described by vector \((a1a2, b1, \Lambda)\). In case this vector is indeed undesirable, some refinement of the component design is required in order to ensure that \((a1a2, b1b2, c1)\) and \((a1a2, b1b3, c1)\) can only be reached through vector \((a1a2, \Lambda, \Lambda)\) excluding any path that would involve vector \((a1a2, b1, \Lambda)\). If on the contrary vector \((a1a2, b1, \Lambda)\) represents reasonable behaviour and such a sequence of calls to operations should be allowed, then our model is detecting it and serves as a designer's aid in finding the complete set of allowed behaviours of the component.

Now based on Figure 4.10, we examine whether the discreteness property holds. That this is so, is best illustrated diagrammatically. By inspection, we have the case depicted as a Hasse diagram in Figure 4.11, in which each subgraph below a given node exhibits the characteristic structure of a lattice. Notice the familiar lozenge shapes.

![Figure 4.11: Discreteness of CMenu component](image)

It might be worth noting that the example shows the interrelationship between concurrency and discreteness. These concepts are closely related and one cannot be considered without the other.

For local left-closure, we concentrate on those vectors in \(V_M\) which have at least one
coordinate containing a sequence of length greater than one and examine their predecessors. Figure 4.12 demonstrates that for each vector in \( V_M \) with at least two events in one of its coordinates there is some other vector in \( V_M \) which has either the same sequence of events, at that specific coordinate, or the same reduced by one event. This implies that the CMem component is locally left-closed.

\[
\begin{array}{cccc}
(a_1a_2, b_1b_2, c_1) & (a_1a_2, b_1b_3, c_1) \\
(a_1a_2, b_1, c_1) & (a_1a_2, b_1, c_1) \\
(a_1a_2, A, c_1) & (a_1a_2, b_1, A) & (a_1, b_1b_2, A) & (a_1, b_1b_3, A) \\
(a_1a_2, A, A) & (a_1a_2, A, A) & (a_1, b_1, A) & (a_1, b_1, A) \\
(a_1, A, A) & (a_1, A, A) & (a_1, A, A) & (a_1, A, A) \\
\end{array}
\]

Figure 4.12: Local left-closure of CMem component

Having established both discreteness and local left-closure for the CMem component, we may now say it is well-behaved.

In this extended example, we formalised a component and established discreteness and local left-closure to make it well-behaved. Obviously, a component designer would not be working at such a level of detail. The examples show how the theory does indeed locate potential design errors. Ultimately, we would like to hide it under design rules / guidelines which would automatically flag up such cases.

In practice sometimes it might be the case that the latency of messages (sensing events) from the remote control is substantially larger than the latency between components embedded in a TV platform. So in our case study, CMem has plenty of time to issue \( c_1 \) before the user can issue \( b_1 \) through the remote control. However, this is not
4.4. Illustration by example

reflected in the sequence diagram of Figure 4.8 and not included in the specification. Such implicit assumptions based on an existing product or system can cause inconsistencies when the component in question is placed in a different context where the assumptions are no longer valid. For instance, consider reusing the CMenu component in a different configuration for a new product with upgraded functionality in which it no longer receives the fine tune request directly through the remote control.

Ideally, feedback should be passed back to the component designer(s) and this feedback should be given again in the form of UML diagrams. For instance, Figure 4.13 shows a sequence diagram indicating the potential pitfalls of the initial description of behaviour for the CMenu component that were identified in the analysis of its behaviour within our formal framework.

Figure 4.13: Forbidden behaviour of the CMenu
We use the neg interaction fragment from UML 2.0 [OMG04] which specifies a forbidden or disallowed sequence of events. The first operand of the alt fragment refers to the case of $b_1$ occurring before $c_1$ while the second operand describes the case where $c_1$ and $b_1$ occur concurrently (after $a_2$).

A similar situation appeared in the case study of Section 3.3 which was concerned with the sequence diagram for a resource allocation telecommunications protocol, although we used the race-free version of Figure 3.9 in that case to demonstrate how the component language can be obtained in the first place. It became apparent in our case studies that it is important to explore the consequences of design decisions and resolve race conditions or at least document any implicit assumptions at the specification level as they often give rise to pathological behaviour.

We have seen that normality, and in particular checking against discreteness, removes inconsistencies that arise as a result of race conditions. In what follows we discuss this connection in more detail.

A race condition in a sequence diagram is possible between ordered locations along a particular component lifeline when one location associated with a send event is followed by a location associated with a receive event, and the two events involve different components with which the component in question interacts (and thus they engage distinct interfaces).

In terms of the formalisation of a UML2.0 sequence diagram given in Section 3.2, we may say that two locations $l, l'$ with $\text{time}(l) = \text{time}(l') - 1$ are in a race condition if there exist $(l, e_1) \in SE$ and $(e_2, l') \in RE$ where $e_1 \in B_1(i'), i' \in I_{\Sigma_1}$ and $e_2 \in B_2(i'')$, $i'' \in I_{\Sigma_2}$, for components $c_1 = (\Sigma_1, V_1), c_2 = (\Sigma_2, V_2)(c_1 \neq c_2 \neq c = (\Sigma, V))$ appearing in the diagram.

The fact that $(l, e_1) \in SE$ means that there is a column vector $g \in E_\Sigma$ such that $g(i') = e_1$ and $g(i) = \Lambda, i \in I_\Sigma \setminus \{i'\}$. Similarly, the fact that $(e_2, l') \in RE$ means that there is a column vector $g' \in E_\Sigma$ such that $g'(i'') = e_2$ and $g'(i) = \Lambda, i \in I_\Sigma \setminus \{i''\}$. This implies that $g \mid \text{ind} g'$ and since they are associated with consecutive locations $l, l'$ they also occur consecutively. So, the events they represent are concurrent.

This means that the resulting behaviours are bounded above by the vector $g$ for which
4.4. Illustration by example

\[ u = u \cdot e \cdot e' = u \cdot e' \cdot e. \] This gives rise to a diamond in the order structure of the corresponding component language \( V \) (Section 4.3.2).

In the sequence diagram, the locations associated with \( e, e' \) are ordered and hence the diagram specifies a causal ordering between \( e, e' \) and therefore captures one ordering explicitly, but does not exclude that \( e, e' \) happen in the reverse order or simultaneously. As a result, it captures one scenario explicitly but allows for inherent scenarios that are possible due to the inherent causal orderings which are not specified in the diagram.

Discreteness requires that whenever two component vectors are bounded above in the language, then their least upper bound and greatest lower bound must be in it. This implies that the column vectors used in building up the two component vectors are independent and commute and thus the events they represent are concurrent. In this way, all possible orderings between the events they represent are considered in the corresponding discrete component language.

It can be seen from the sequence diagram of Figure 4.8 and the analysis in our case study that the nondeterministic choice between \( b1 \) and \( a2 \) is the source of the race condition between \( b1 \) and \( c1 \). The events \( b1, a2 \) lead to the same behaviour \( (a2a2, b1, c1) \) and thus there is no way to impose in the sequence diagram that \( c1 \) occurs immediately after \( a2 \) and nothing is allowed to happen in between (which is the product requirement) because \( b1 \) can potentially happen concurrently with both of them.

In considering discreteness of a component language in our formal framework, we uncover inherent orderings among component vectors. The behaviours these vectors describe, give rise to inherent scenarios that are not specified in the sequence diagram but are possible. Such inherent scenarios are the cause of pathological behaviour either at the present configuration or when the component is placed in a different configuration and 'carries' these inherent scenarios with it while its correct participation in the new configuration is only checked against the explicit scenarios specified in the accompanying scenario-based specification.
4.5 Concluding note

In this chapter, we presented the basic properties of a component language. These were given in terms of the order theoretic properties of component vectors. We also examined how the various relationships between events (calls to operations) on component interfaces manifest themselves in the order structure of the corresponding component language. A key observation was that the ordering between events on different component interfaces is determined by what other vectors are in the language - it is dependent on context. Hence, a component language comes equipped with an order structure that indicates possible constraints on the order in which operations of the component can or should be called.

For instance, consider the component language,

$$V_1 = \{(A, A), (a, A), (A, b), (a, b)\}$$

which is discrete and locally left-closed. The events $a, b$ in $V_1$ are concurrent (the component vectors form a diamond). Next, consider the component language,

$$V_2 = \{(A, A), (a, b)\}$$

which is discrete and locally left-closed. The events $a, b$ in $V_2$ are simultaneous. Finally, consider the component language

$$V_3 = \{(A, A), (a, A), (A, b)\}$$

which is discrete and locally left-closed. The events $a, b$ in $V_3$ are mutually exclusive. It can be seen that it is possible within our framework to formally capture phenomena such as nondeterminism and concurrency, as well as simultaneity, on interfaces of the same component.

Further, by exploiting the basic order theoretic properties of component vectors, we identified two additional properties, discreteness and local left-closure, which determine the so-called well-behaved components. This allowed for a straightforward representation of every event occurrence on a component interface. It is also noteworthy that well-behavedness is preserved under composition of components. This will be discussed in the next chapter (Chapter 5).
4.5. Concluding note

Discreteness and local left-closure can have practical benefits for component-based design in that they determine whether the set of behaviours is complete or on the contrary possible, and potentially pathological, behaviours have been omitted. Missing behaviours may be the result of inherent scenarios in sequence diagrams that are hidden under race conditions for example, or behind implicit assumptions based on embedding the component in a particular environment, which may no longer be valid when the component is placed in a slightly different configuration.

Such inconsistencies can be avoided in well-behaved components and this was demonstrated in Section 4.4 where we also hinted towards providing component designer(s) with feedback on pathological behaviours, that often go unnoticed under human inspections, in the form of UML sequence diagrams.

Apart from identifying missing behaviours, well-behavedness has as a consequence that the component language can be associated with:

- a discrete behavioural presentation, which provides an event-based model for component behaviour (see Chapter 6 for this association)

- a class of automata, the so-called $\Sigma$-automata, which provide a state-based model for component behaviour (see Chapter 7 for this association).
Chapter 4. Properties of a Component Language
Chapter 5

Composition of Components

The main interest in the study of components has to do with understanding the consequences of putting components together to form a system or provide some (additional) functionality within a system. Current component technologies such as the OMG’s CORBA Component Model [OMG02], Microsoft’s COM / .NET [Cor] and Sun’s EJB [Mic03] support the rapid assembly of systems from pre-fabricated components. However, there is little, if any, support for reasoning about the resulting system until its parts have been combined, executed and tested as a whole. A prerequisite for the predictable assembly of component-based systems is the ability to formally reason about the behaviour of the composite based on properties of the individual components. In this chapter, we are primarily concerned with this issue.

We start by giving a formal notion of composition of components. The basic idea is that composition can only take place if a required interface of one component can be matched to a provided interface of another. This implies certain conditions on the corresponding signatures and component languages. In the previous chapter, we considered constraints on a component language that characterised well-behaved components. In this chapter, we examine the effect of composition on the participating components with respect to well-behavedness, which amounts to preservation of discreteness and local left-closure in component languages under composition.

Our study of composition of components is outlined in [MS03] while [MS04b] contains the complete proofs and establishes the algebraic properties of composition; it is com-
mutative and associative. Apart from the mathematical development that establishes discreteness of the resulting composite component language, we also advocated the use of Hasse diagrams as a diagrammatic representation of well-behavedness. This was used in performing the case study appearing in [MS04b] and suggests a more intuitive way of checking against discreteness and local left-closure in component languages.

In this chapter we outline this work and propose extensions to the theory underlying component composition. In particular, we extend the notion of consistent signatures to allow for more than strict interface typing (Lemma 5.1.1) and show that we can omit the third compatibility condition given in [MS04b] by way of considering the more general interaction between consistency, composition and least upper bounds of component vectors (Lemma 5.3.3). We also discuss the use of structure diagrams in UML2.0 [OMG04] as a graphical notation for the formal notion of component composition in our approach. We demonstrate the ideas through the extended example of Chapter 4.

5.1 Formalisation of Component Composition

Components are put together by connecting (binding in Koala terminology or wiring in UML dialect) provided interfaces of one component with required interfaces of the other(s). A prerequisite is that, for a pair of connected interfaces, the requests issued through the required interface are accepted by the provided interface.

In our approach, this comes down to checking whether the component vectors of each component agree on the respective coordinates corresponding to the connected interfaces. The idea is that, if component $c_1$ provides interface $i$ and component $c_2$ requires interface $i$, then a behaviour of $c_1$ and a behaviour of $c_2$ (as described by the corresponding component vectors) can only be composed if their restrictions to interface $i$ are the same.

This notion of composition builds on the well-known concept of parallel composition found in process algebras such as CCS [Mil80] and CSP [Hoa85]. In particular, the way the sequences (one from each component) associated with the connected interface $i$ are combined is reminiscent of the interface parallel operator described in [Sch00], which
requires synchronisation only on those events appearing in a common interface between processes.

The difference is that synchronisation in our framework is slightly more complex. An event (operation call) associated with the required interface of one component must also be associated with the provided interface of the other. Communication between the two involves the occurrence of this event on the required interface of one component (experienced as making an operation call or sending a signal) and the occurrence of the same event on the provided interface of the other component (experienced as accepting an operation call or receiving a signal). In order to put two components together, we require that such synchronised event occurrences respect the ordering prescribed in the sequence of events corresponding to the connected interface.

This notion of composition covers models with zero delay communication like the STATEMATE tool [HP96] from i-Logix. In this case, the send and receive events must be simultaneous, in the sense of simultaneity considered in our formal framework. For delayed communication (synchronous or asynchronous) where the send and receive events are not considered simultaneous, we need to make the assumption that the send events are received in the same order, in addition to the usual condition that the send event precedes the corresponding receive event.

In a fashion similar to the formalisation of a single component, we study the static characteristics (structure) of the resulting composite component and its dynamics (behaviour). In what follows, we describe how signatures and languages of individual components are composed within our formal framework.

### 5.1.1 Composing signatures

We have seen (Definition 3.1.1) that a component is identified by a signature which comprises its sets of provided and required interfaces and also specifies the set of events associated with each interface. Components are composed by connecting interfaces that are required by one component with interfaces provided by another. In addition, the respective sets of events must be the same. This leads to a notion of consistency between the corresponding signatures.
Definition 5.1.1. Suppose that $\Sigma_1 = (P_1, R_1, \beta_1)$, $\Sigma_2 = (P_2, R_2, \beta_2)$ are signatures. We say that $\Sigma_1$ and $\Sigma_2$ are consistent, and we write $\Sigma_1 \downarrow \Sigma_2$, if and only if

- $P_1 \cap P_2 = \emptyset$
- $R_1 \cap R_2 = \emptyset$
- $\forall i \in I_{\Sigma_1} \cap I_{\Sigma_2} : \beta_1(i) = \beta_2(i)$

Suppose that $c_1 = (\Sigma_1, V_1), c_2 = (\Sigma_2, V_2)$ are components. Then, $c_1$ and $c_2$ are consistent, and we write $c_1 \downarrow c_2$, if $\Sigma_1$ and $\Sigma_2$ are consistent.

This is used to determine component interfaces that are eligible for composition. Consistency requires that connected interfaces can only be interfaces provided by one component and required by the other. In mathematical terms,

$$I_{\Sigma_1} \cap I_{\Sigma_2} = (P_1 \cap R_2) \cup (R_1 \cap P_2)$$

Further, consistency ensures that the component developers do not change the component interface at configuration time to suit only one implementation. This is expressed in (3) of Definition 5.1.1 which imposes strict interface typing - the set of operations on each interface must be the same.

However, an interface can be required or provided by more than one component. For instance, both a European frequency-based television and a USA frequency-based television front end can be connected to both a high-end and an economical tuner so long as the tuners support the same interface. With strict interface typing, a tuner providing the new high-end interface cannot be connected to a front end requiring the old economical interface - without adding glue code.

To address such situations, which appear to be a common design pattern in CE products [vO02a], it would be desirable to consider connections between interfaces even when their sets of operations are not exactly the same. It turns out that this is possible, providing that the provided interface contains at least all the functions of the required interface. This is established in the following lemma.

Before giving the lemma, we define what is meant by a decomposition of a signature.
5.1. Formalisation of Component Composition

Definition 5.1.2. Suppose that \( \Sigma = (P, R, \beta) \) is a signature. We say that \( \Sigma' = (P', R', \beta') \) is a decomposition of \( \Sigma \) if \( \Sigma' \) is a signature and \( I_\Sigma \neq I_{\Sigma'} \) and \( O_{P_\Sigma} = O_{P_{\Sigma'}} \).

Thus, a decomposition of a signature is associated with the same set of operations but has a different set of interfaces. We may now give the lemma, which allows to consider connections between interfaces even when their sets of operations, as given by the initial signatures, are not the same.

Lemma 5.1.1. Suppose that \( \Sigma_1 = (P_1, R_1, \beta_1) \) and \( \Sigma_2 = (P_2, R_2, \beta_2) \) are signatures which satisfy (1) and (2) of Definition 5.1.1 but \( \beta_1(i) \neq \beta_2(i) \), some \( i \in I_{\Sigma_1} \cap I_{\Sigma_2} \). If the set of operations associated with \( i \) by the signature in which \( i \) is a provided interface contains at least all the operations in the set of operations associated with \( i \) by the signature in which \( i \) is a required interface, then there is a decomposition \( \Sigma' \) of the signature providing \( i \) such that \( \Sigma' \) is consistent with the signature requiring interface \( i \).

Proof.

Let \( i \in I_{\Sigma_1} \cap I_{\Sigma_2} \). Without loss of generality, suppose that \( i \in R_1 \cap R_2 \). We have \( \beta_2(i) \subseteq \beta_1(i) \).

For \( i_1 \notin I_{\Sigma_1} \cup I_{\Sigma_2} \), define \( P'_1 = P_1 \cup \{i_1\} \), \( R'_1 = R_1 \) and \( \beta'_1 : P'_1 \cup R'_1 \to \varphi(O_{P_{\Sigma_1}}) \) such that \( \beta'_1(i_1) = \beta_1(i_1) \) and \( \beta'_1(i) = \beta_1(i) \setminus \beta_2(i) \).

Let \( \Sigma'_1 = (P'_1, R'_1, \beta'_1) \). Before we show that \( \Sigma'_1 \downarrow \Sigma_2 \), we need to show that \( \Sigma'_1 \) is a signature.

- Since \( i_1 \notin I_{\Sigma_1} \) we have \( i_1 \notin P_1 \) and hence, \( P'_1 \cap R'_1 = (P_1 \cup \{i_1\}) \cap R_1 = \emptyset \).
- We have \( I_{\Sigma'_1} = I_{\Sigma_1} \cup \{i_1\} \). This implies that

\[
\bigcup_{i \in I_{\Sigma'_1}} \beta'_1(i) = \left( \bigcup_{i \in I_{\Sigma_1}} \beta'_1(i) \right) \cup \beta'_1(i_1) \\
= (O_{P_{\Sigma_1}} \setminus (\beta_1(i) \setminus \beta_2(i))) \cup \beta'_1(i_1) \\
= O_{P_{\Sigma_1}} \\
= O_{P_{\Sigma_1}}
\]

\( O_{P_{\Sigma_1}} \setminus (\beta_1(i) \setminus \beta_2(i)) \subseteq O_{P_{\Sigma_1}} \setminus \beta_2(i) \subseteq O_{P_{\Sigma_1}} \cap O_{P_{\Sigma_2}} = O_{P_{\Sigma_1}} \cap O_{P_{\Sigma_2}} = O_{P_{\Sigma_1}} \)
Thus, $\Sigma'_i = (P'_i, R'_i, \beta'_i)$ is a signature, by Definition 3.1.1, which is a decomposition of $\Sigma_i = (P_i, R_i, \beta_i)$.

It remains to show that $\Sigma'_1, \Sigma_2$ are consistent. We have,

- $P'_1 \cap P_2 = (P_1 \cup \{i_1\}) \cap P_2 = \emptyset$ by the fact that $i_1 \notin I_{\Sigma_1} \cup I_{\Sigma_2}$ and hence $i_1 \notin P_2$
- $R'_1 \cap R_2 = R_1 \cap R_2 = \emptyset$ by consistency of $\Sigma_1, \Sigma_2$
- $\forall i \in I_{\Sigma_1} \cap I_{\Sigma_2} : \beta'_1(i) = \beta_2(i)$, by definition of $\beta'_1$ above

Hence, by Definition 5.1.1 we have $\Sigma'_1 \downarrow \Sigma_2$ which completes the proof. □

It can be seen that the above lemma allows for some flexibility, with regard to (3) of Definition 5.1.1, for determining component interfaces that are eligible for composition. It says that if the provided interface contains at least all the operations of the required interface, then we can obtain a decomposition of the signature providing the interface such that it is consistent with the signature requiring the interface. The signature that arises as the decomposition, introduces a new provided interface which is assigned the superfluous operations of the old provided interface (in relation to those of the required interface in the initial signatures).

We are now set to consider the static characteristics of the resulting composite component. These are given in terms of the composition of the corresponding (consistent) signatures.

**Definition 5.1.3.** Suppose that $\Sigma_1 = (P_1, R_1, \beta_1)$ and $\Sigma_2 = (P_2, R_2, \beta_2)$ are consistent signatures. Define $\Sigma_1 \oplus \Sigma_2 = \Sigma$ where,

- $P = (P_1 \cup P_2) \setminus (R_1 \cup R_2)$
- $R = (R_1 \cup R_2) \setminus (P_1 \cup P_2)$
- $\beta(i) = \beta_j(i)$ wherever $i \in I_{\Sigma_j}, j = 1, 2$ (recall that $I_{\Sigma_j} = P_j \cup R_j$ by Definition 3.1.1)

The composite of two signatures, as given by the above definition, is shown to be itself a signature in the following lemma.
5.1. Formalisation of Component Composition

Lemma 5.1.2. Suppose that $\Sigma_1, \Sigma_2$ are consistent signatures, then $\Sigma_1 \oplus \Sigma_2$ is a signature.

Proof.

We first prove that $\beta$ is a well defined function. Since $I_\Sigma \subseteq I_{\Sigma_1} \cup I_{\Sigma_2}$, it suffices to show that if $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$ then $\beta_1(i) = \beta_2(i)$ which is precisely point (3) of Definition 5.1.1. Finally, we note that

$$P \cap R = ((P_1 \cup P_2) \setminus (R_1 \cup R_2)) \cap ((R_1 \cup R_2) \setminus (P_1 \cup P_2)) = \emptyset$$

which completes the proof. \(\square\)

It can be seen that the composite signature $\Sigma_1 \oplus \Sigma_2$ is formed from the signatures of the individual components by eliminating all interfaces participating in the ensuing communication. In effect, the composite signature internalises all connected interfaces. Its provided interfaces then are those remaining provided interfaces of each component and its required interfaces are those remaining required interfaces of each component.

5.1.2 Composing component vectors

Up to this point we have been concerned with the static characteristics of the resulting composite. We now turn our attention to the dynamic characteristics of the composite.

In composing signatures we required (Definition 5.1.1) that the respective sets of operations associated with the common interface are the same, or according to Lemma 5.1.1 the provided interface contains at least all operations of the required interface. In composing component vectors, we also want to ensure that the respective sequences of operation calls on the common interface are the same. The idea behind this requirement is the following.

In any behaviour of the composite system, each component $c_j, j = 1, 2$, will have engaged in a piece of behaviour described by $v_j$. If $i$ is an interface common to both $c_1$ and $c_2$, then it will be a provided interface of one and a required interface of the other. Without loss of generality, suppose that it is a required interface of $c_1$ and a provided interface of $c_2$. Then, $v_j(i)$ represents the sequence of calls to operations made from $c_1$.
to $c_2$ through interface $i$, which is precisely the sequence of operation calls $y_2(t)$. This
leads to a notion of consistency among component vectors, which determines pairs of
component vectors (one from each component) that can be composed.

$X \triangle Y$ is used to denote the symmetric difference of the sets $X, Y$ and is defined to be
$(X \setminus Y) \cup (Y \setminus X)$.

**Definition 5.1.4.** Let $c_1 = (\Sigma_1, V_1)$ and $c_2 = (\Sigma_2, V_2)$ be components and suppose that
$\Sigma_1 \downarrow \Sigma_2$. We say that $u_1 \in V_1$ and $u_2 \in V_2$ are consistent, and we write $u_1 \downarrow u_2$, if

$$u_1 \mid_{I_{\Sigma_1} \cap I_{\Sigma_2}} = u_2 \mid_{I_{\Sigma_1} \cap I_{\Sigma_2}}$$

where if $f$ is a function, $f \mid_X$ denotes the restriction of function $f$ to the set $X$, in
which case we define,

$$u_1 \oplus u_2 = (u_1 \cup u_2) \mid_{I_{\Sigma_1} \Delta I_{\Sigma_2}}$$

where $u_1 \cup u_2 : I_{\Sigma_1} \Delta I_{\Sigma_2}$ satisfies

$$(u_1 \cup u_2)(i) = \begin{cases} u_1(i), & i \in I_{\Sigma_1} \\ u_2(i), & i \in I_{\Sigma_2} \end{cases}$$

which is well defined if $u_1 \downarrow u_2$.

Recall that component vectors were defined to be functions (see Definition 3.1.2), and
thus $u_1 \mid_{I_{\Sigma_1} \cap I_{\Sigma_2}}$ is the vector obtained from $u_1 \in V_1$ by removing all the coordinates
except those that correspond to interfaces of $c_1$ that are connected to interfaces of $c_2$.
Similarly, for $u_2 \mid_{I_{\Sigma_1} \cap I_{\Sigma_2}}$ and $c_2$.

It might be worth mentioning that if $f_1, f_2$ are functions with $f_1 : A_1 \rightarrow B_1$ and $f_2 : A_2 \rightarrow B_2$, then, in general, $f_1 \cup f_2$ is a relation such that $f_1 \cup f_2 \subseteq (A_1 \cup A_2) \times (B_1 \cup B_2)$
given by $(a, b) \in f_1 \cup f_2 \iff f_1(a) = b \lor f_2(a) = b$. Also, if $f_1 \mid_{A_1 \cap A_2} = f_2 \mid_{A_1 \cap A_2}$ (which
in the notation of Definition 5.1.4 is $f_1 \downarrow f_2$), then $f_1 \cup f_2$ is a function. Consequently,
in the case of component vectors, $u_1 \oplus u_2$ is always a function since it is only defined
for component vectors $u_1, u_2$ which agree on common coordinates, i.e. for $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$.

Furthermore, it is a consequence of the above definition that if $\Sigma_1, \Sigma_2$ are signatures
such that $\Sigma_1 \downarrow \Sigma_2$, and $u_1 \in V_1$ and $u_2 \in V_2$ such that $u_1 \downarrow u_2$, then $u_1 \oplus u_2 \in V_{\Sigma_1 \oplus \Sigma_2}$.
5.1. Formalisation of Component Composition

The following remark is also a consequence of Definition 5.1.4 and will be used in the sequel, specifically with reference to Lemma 5.3.1. It says that if a component vector in one language describes earlier behaviour than a vector in the same language, and this is also the case for their consistent counterparts in the other language, then the composite component vectors resulting from the composition of the consistent pairs preserve this ordering in the composite language.

Remark 5.1.1. If \( c_1 = (\Sigma_1, V_1), c_2 = (\Sigma_2, V_2) \) are consistent components and \( y_1, y_1 \in V_1 \) and \( y_2, y_2 \in V_2 \) such that

1. \( y_1 \subseteq y_2 \) and \( y_1 \subseteq y_2 \)
2. \( y_1 \leq y_1 \) and \( y_2 \leq y_2 \)

then, \( y_1 \oplus y_2 \leq y_1 \oplus y_2 \).

Proof.

Let \( i \in I_{\Sigma_2} \setminus I_{\Sigma_3} \). Then, by Definition 5.1.4 \( (y_1 \oplus y_2)(i) = y_1(i) \) and \( (y_2 \oplus y_2)(i) = y_2(i) \). Since \( y_1 \subseteq y_1 \) we can deduce that \( (y_1 \oplus y_2)(i) \leq (y_1 \oplus y_2)(i) \).

Similarly, when \( i \in I_{\Sigma_2} \setminus I_{\Sigma_3} \) we have \( (y_1 \oplus y_2)(i) = y_2(i) \) and \( (y_1 \oplus y_2)(i) = y_2(i) \). Since \( y_2 \leq y_2 \), we conclude that \( (y_1 \oplus y_2)(i) \leq (y_1 \oplus y_2)(i) \).

Hence, it follows that \( y_1 \oplus y_2 \leq y_1 \oplus y_2 \). \( \square \)

Now, we can give a formal definition of composition of components.

Definition 5.1.5. Suppose that \( c_1 = (\Sigma_1, V_1) \) and \( c_2 = (\Sigma_2, V_2) \) are consistent components. Then, we define \( c_1 \oplus c_2 = (\Sigma, V) \) where,

- \( \Sigma = \Sigma_1 \oplus \Sigma_2 \)
- \( V = V_1 \oplus V_2 \) where \( V_1 \oplus V_2 = \{ u \in V_2 | \exists u_1 \in V_1, \exists u_2 \in V_2 : u_1 \downarrow u_2 \land u = u_1 \oplus u_2 \} \)

Thus, the signature \( \Sigma \) of the composite is given by Definition 5.1.3 and the language of the composite consists of component vectors formed over \( \Sigma \), which comprise vectors (from each component language) that are consistent (Definition 5.1.4).
It is straightforward to show that \( c_1 \oplus c_2 = (\Sigma, V) \) is a component whenever \( c_1, c_2 \) are consistent components.

**Remark 5.1.2.** Suppose that \( c_1 = (\Sigma_1, V_1) \) and \( c_2 = (\Sigma_2, V_2) \) are consistent components. Then, in the notation of Definition 5.1.5, \( c_1 \oplus c_2 = (\Sigma, V) \) is a component.

**Proof.**

\( \Sigma \) is a sort by Lemma 5.1.2 and \( V \subseteq V_2 \) holds by definition. Hence, \( c_1 \oplus c_2 \) is a component by Definition 3.1.3. □

### 5.1.3 Algebraic properties of composition

We have seen that the operation of composition in our approach takes two components and produces a composite component. In the component-oriented paradigm to software engineering, systems are constructed by putting together a number of pre-built components. This means that the resulting composite needs to be further composed with other components or other composites. To ensure that this is done in a principled way, the operation of composition has to satisfy certain properties.

In this section, we are concerned with the algebraic properties of composition. In particular, we show that the operation \( \oplus \) is commutative and associative.

The following lemma establishes commutativity.

**Lemma 5.1.3.** Suppose that \( c_1, c_2 \) are components, then \( c_1 \updownarrow c_2 \) if and only if \( c_2 \updownarrow c_1 \), and in either case \( c_1 \oplus c_2 = c_2 \oplus c_1 \).

**Proof.**

Definitions 5.1.1, 5.1.3, 5.1.4, 5.1.5 are all symmetric on \( \Sigma_2, \Sigma_2 \). Thus, \( c_1 \downarrow \Sigma_2 \) implies \( c_2 \downarrow \Sigma_1 \) in which case \( \Sigma_1 \downarrow \Sigma_1 = \Sigma_2 \oplus \Sigma_1 \). By Definition 5.1.1, the definitions are all symmetric on \( c_1, c_2 \) and thus the result also applies to \( c_1, c_2 \). □

We now turn our attention to associativity. First we establish conditions under which \( (c_1 \oplus c_2) \oplus c_3 \) and \( c_1 \oplus (c_2 \oplus c_3) \) are defined. Then we show that they are equal.

We start with a lemma that describes the basic interaction between '\( \downarrow \)' and '\( \oplus \)'.

Lemma 5.1.4. Suppose that $c_1, c_2, c_3$ are components such that $c_j \downarrow c_k$ when $j \neq k$. Then, $c_1 \downarrow (c_2 \oplus c_3)$ and $(c_1 \oplus c_2) \downarrow c_3$.

Proof.

If the first claim $c_1 \downarrow (c_2 \oplus c_3)$ were true, then by interchanging the roles of the $c_j$ we would have $c_3 \downarrow (c_1 \oplus c_2)$. By Lemma 5.1.3 then, we have $(c_1 \oplus c_2) \downarrow c_3$ which is our second claim. Thus, it suffices to prove our first claim. Let $c = (\Sigma, V) = c_2 \oplus c_3$. Checking against Definition 5.1.1 for consistency of $c_1 \downarrow c$ we have,

$$P_1 \cap P = P_1 \cap \left((P_2 \cup P_3) \setminus (R_2 \cup R_3)\right)$$

$$\subseteq P_1 \cap (P_2 \cup P_3)$$

$$= (P_1 \cap P_2) \cup (P_1 \cap P_3)$$

$$= \emptyset$$

which is precisely the point of (1) of Definition 5.1.1. In similar fashion,

$$R_1 \cap R = R_1 \cap \left((R_2 \cup R_3) \setminus (P_2 \cup P_3)\right)$$

$$\subseteq R_1 \cap (R_2 \cup R_3)$$

$$= (R_1 \cap R_2) \cup (R_1 \cap R_3)$$

$$= \emptyset$$

which is precisely the point of (2) of Definition 5.1.1. Finally, suppose that $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$. We have two cases: either $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$ in which case $\beta_{\Sigma_1}(i) = \beta_{\Sigma_2}(i) = \beta_{\Sigma}(i)$, or $i \in I_{\Sigma_1} \cap I_{\Sigma_3}$ in which case $\beta_{\Sigma_1}(i) = \beta_{\Sigma_3}(i) = \beta_{\Sigma}(i)$. We have proved that $\Sigma_1 \downarrow \Sigma$. Hence, $\Sigma_1 \downarrow (\Sigma_2 \oplus \Sigma_3)$. Now Definition 5.1.1 gives $c_1 \downarrow (c_2 \oplus c_3)$ which completes the proof. □

Remark 5.1.3. Suppose that $\Sigma_1, \Sigma_2, \Sigma_3$ are signatures such that $\Sigma_j \downarrow \Sigma_k$ when $j \neq k$, then $I_{\Sigma_1} \cap I_{\Sigma_2} \cap I_{\Sigma_3} = \emptyset$.

Proof.

Suppose that $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$. Without loss of generality, and in view of Definition 5.1.1, we may assume that $i \in P_1 \cap R_2$. Now, $i \notin R_3$ as $\Sigma_2 \downarrow \Sigma_3$, and $i \notin P_3$ as $\Sigma_4 \downarrow \Sigma_3$. 


Similarly, if we assume that $i \in R_1 \cap P_2$, we may deduce that $i \not\in R_3$ and $i \not\in P_3$. Hence, $i \not\in I_{E_3}$, which completes the proof. □

Associativity of $\oplus$ on components can now be considered in terms of the corresponding signatures and component languages. The following lemma is concerned with the signatures part while Proposition 5.1.1 is concerned with the languages part and establishes associativity of $\oplus$ on components by bringing everything together.

Lemma 5.1.5. Suppose that $\Sigma_1, \Sigma_2, \Sigma_3$ are signatures such that $\Sigma_j \downarrow \Sigma_k$ when $j \neq k$, then

$$((\Sigma_1 \oplus \Sigma_2) \oplus \Sigma_3) = (P, R, \beta) = \Sigma_1 \oplus (\Sigma_2 \oplus \Sigma_3)$$

where

- $P = (P_1 \cup P_2 \cup P_3) \setminus (R_1 \cup R_2 \cup R_3)$
- $R = (R_1 \cup R_2 \cup R_3) \setminus (P_1 \cup P_2 \cup P_3)$
- $\beta : P \cup R \to p(\text{Op}_S)$ is given by $\beta(i) = \beta_j(i)$, whenever $i \in I_{\Sigma_j}$

Proof.

Since $\Sigma_j \downarrow \Sigma_k$, $j \neq k$, we have that $(\Sigma_1 \oplus \Sigma_2) \downarrow \Sigma_3$ is defined by Lemma 5.1.4. Thus, in view of Definition 5.1.3, $(\Sigma_1 \oplus \Sigma_2) \oplus \Sigma_3$ is defined. Also, again since the signatures are pairwise consistent, we have by Definition 5.1.1 that $\beta_j(i) = \beta_k(i)$ whenever $i \in I_{\Sigma_j} \cap I_{\Sigma_k}$, so $\beta$ is well defined.

Let $\Sigma'' = \Sigma_1 \oplus \Sigma_2$ and $\Sigma' = \Sigma'' \oplus \Sigma_3$. We must show that $\Sigma' = (P', R', \beta') = (P, R, \beta)$.

First we show that $P = P'$.

Suppose that $i \in P'$. Then, $i \in (P'' \cup P_3) \setminus (R'' \cup R_3)$ which means that $i \in P'' \cup P_3$ and $i \not\in R'' \cup R_3$. We consider two cases: $i \in P''$ and $i \in P_3$.

- If $i \in P''$, then we have $i \in P_1 \cup P_2$ and $i \not\in R_1 \cup R_2$. Since we also have $i \not\in R_3$, we may deduce that $i \in (P_1 \cup P_2) \setminus (R_1 \cup R_2 \cup R_3) \subseteq P$. This implies that $P'' \subseteq P$.

- If $i \in P_3$, then $i \not\in R_3$. Hence, $i \in P_3 \setminus R_3$. Since $i \not\in R'' \cup R_3$ we may deduce that $i \in P_3 \setminus (R'' \cup R_3) = P_3 \setminus (R_1 \cup R_2 \cup R_3) \subseteq P$. This implies that $P_3 \subseteq P$. 


We have shown that \( P' \subseteq P \).

Conversely, suppose that \( i \in P \). Then, \( i \in ((P_1 \cup P_2) \cup P_3) \setminus ((R_1 \cup R_2) \cup R_3) \) which means that \( i \in P'' \setminus P_3 = P_1 \cup P_2 \cup P_3 \) and \( i \notin R_1 \cup R_2 \cup R_3 \). Again, we consider two cases: \( i \in P'' \) and \( i \in P_3 \).

- If \( i \in P'' \), then \( i \in P'' \cup P_3 \). Since \( i \notin R_1 \cup R_2 \cup R_3 = R'' \cup R_3 \), we may deduce that \( i \in (P'' \cup P_3) \setminus (R'' \cup R_3) \). But \( (P'' \cup P_3) \setminus (R'' \cup R_3) = P' \). Hence, \( i \in P' \). This implies that \( P'' \subseteq P' \).

- If \( i \in P_3 \), then \( i \notin R_3 \). Hence, \( i \in P_3 \setminus R_3 \). Since \( i \notin R'' \cup R_3 \), we may deduce that \( i \in P_3 \setminus (R'' \cup R_3) = (P_3 \setminus (R_1 \cup R_2 \cup R_3)) \subseteq P' \). Hence, \( i \in P' \). This implies that \( P_3 \subseteq P' \).

We have now shown that in both cases, \( i \in P' \). Hence, \( P \subseteq P' \). We have also seen that \( P'' \subseteq P \). Thus, \( P = P' \). Similarly, and exchanging the roles of \( P \) and \( R \), we have \( R = R' \).

Finally, we note that \( \beta(i) = \beta_j(i) = \beta'(i) \) whenever \( i \in I_{\Sigma_j} \). Thus, we have shown that \( (\Sigma_j \oplus \Sigma_3) \oplus \Sigma_3 = (P, R, \beta) = (\Sigma_2 \oplus \Sigma_3) \oplus \Sigma_1 \) (the right part by interchanging the roles of the \( \Sigma_1 \)). From Lemma 5.1.3 it can be concluded that \( (\Sigma_2 \oplus \Sigma_3) \oplus \Sigma_1 = \Sigma_1 \oplus (\Sigma_2 \oplus \Sigma_3) \).

Thus, we now have the claim of the lemma, \( (\Sigma_1 \oplus \Sigma_2) \oplus \Sigma_3 = (P, R, \beta) = \Sigma_1 \oplus (\Sigma_2 \oplus \Sigma_3) \).

Proposition 5.1.1. Suppose that \( c_1, c_2, c_3 \) are components such that \( c_j \downarrow c_k \) when \( j \neq k \), then \( c_1 \oplus c_2 \oplus c_3 \) and \( c_1 \oplus (c_2 \oplus c_3) \) are both defined and equal.

Proof.

Both are defined by Lemma 5.1.4. Let \( c = (\Sigma, V) = c_1 \oplus c_2 \), \( c = c_1 \oplus c_3 \) and let \( c' = (\Sigma', V') = c_2 \oplus c_3 \), \( c' = c_1 \oplus c' \). We must show that \( c = c' \). We have \( \Sigma = \Sigma' \) by Lemma 5.1.5. Hence, we must show that \( \tilde{V} = \tilde{V}' \).

Let \( y \in \tilde{V} \), then there exists \( u_j \in V_j, j = 1, 2, 3 \) such that

\[
\begin{align*}
&u_1 \downarrow u_2 \\
&(u_1 \oplus u_2) \downarrow u_3 \\
&\tilde{V} = (u_1 \oplus u_2) \oplus u_3
\end{align*}
\]
We shall prove that

\[ u_2 \downarrow u_3 \]
\[ u_1 \downarrow (u_2 \oplus u_3) \]
\[ u = u_1 \oplus (u_2 \oplus u_3) \]

Suppose that \( i \in I_{\Sigma_2} \cap I_{\Sigma_3} \). Then \( i \notin I_{\Sigma_1} \), by Remark 5.1.3. Thus, we now have that \( i \in I_{\Sigma_2} \setminus I_{\Sigma_1} \subseteq (I_{\Sigma_2} \setminus I_{\Sigma_1}) \cup (I_{\Sigma_3} \setminus I_{\Sigma_1}) = I_{\Sigma_2} \setminus I_{\Sigma_1} = I_{\Sigma_2} \setminus I_{\Sigma_3} = I_{\Sigma} \). Hence, \( i \in I_{\Sigma_2} \cap I_{\Sigma_3} \) and now, \( u_2(i) = (u_1 \oplus u_2)(i) \). Since \( (u_1 \oplus u_2) \downarrow u_3 \) and \( i \in I_{\Sigma_2} \) and \( i \in I_{\Sigma_3} \) we have \( u_3(i) = (u_1 \oplus u_2)(i) \). Hence, \( u_2(i) = u_3(i) \) and we have proved that \( u_2 \downarrow u_3 \).

Let \( i \in I_{\Sigma_2} \cap I_{\Sigma_3} \). As \( I_{\Sigma_2} \cap I_{\Sigma_3} = I_{\Sigma_2} \cap (I_{\Sigma_3} \setminus I_{\Sigma_2}) \), either \( i \in I_{\Sigma_2} \cap I_{\Sigma_3} \) or \( i \notin I_{\Sigma_2} \cap I_{\Sigma_3} \).

In the first case, \( (u_2 \oplus u_3)(i) = u_3(i) = u_3(i) \), and in the second, \( (u_2 \oplus u_3)(i) = u_3(i) = (u_1 \oplus u_2)(i) = u_1(i) \). Hence, we have shown that \( u_1 \downarrow (u_2 \oplus u_3) \).

Finally, \((u_1 \oplus u_2) \oplus u_3)(i) \) and \((u_1 \oplus (u_2 \oplus u_3))(i) \) are both equal to \( u_j(i) \) where \( j \) is the unique number such that \( i \in I_{\Sigma_j} \) (\( j \) is unique by Remark 5.1.3). Thus, we have shown that \( \bar{v} = \bar{v}' \) which completes the proof. \( \square \)

Therefore, we have shown that the operation \( \oplus \) of composition of components is commutative and associative. This means that the composite of two components can be further composed with other components or other composites to form a larger system.

### 5.2 Visualising Composition with UML

In this section, we hint towards the use of UML notation for a diagrammatic description of component composition. Specifically, UML2.0 contains composite structure diagrams which can be useful for visualising the underlying formal concepts of component composition.

Figure 5.1 depicts the composition of the CMenu and CTimer components of the case study given in Section 4.4 using the notation of structure diagrams in UML 2.0 [OMG04]. We make use of the connector concept in UML 2.0 (see section 8.3.2, pp. 160-163 in the UML 2.0 specification document [OMG04]).
An *assembly* connector between two components in UML defines that one component provides the services that the other component requires. The UML 2.0 specification imposes the constraint that an assembly connector can only be defined from a required interface to a provided interface. This constraint is formally captured in our model by the definition of consistent signatures, Definition 5.1.1, which dictates that connected interfaces belong to the set \( I_{\Sigma_1} \cap I_{\Sigma_2} = (P_1 \cap R_2) \cup (P_2 \cap R_1) \).

The semantics of the assembly connector, given in the UML 2.0 specification, is that signals (operation requests or events) originate in a required interface, travel along an instance of the connector and are delivered to a provided interface. In our component model the dynamics of composition (Definition 5.1.4) formalises that and further, imposes that the sequence of operation calls experienced at the required interface is the
same as that experienced on the provided interface. Moreover, the definition entails that the ordering of events is preserved on the target interface, something which is also in the spirit of the UML assembly connector which is seen as simply providing the connection between the two.

The assembly connector is represented using the 'ball and socket' notation (see interface IDetectSignal in Figure 5.1).

We also use another kind of connector in the structure diagram of Figure 5.1. A delegation connector in UML 2.0 is a connector that links the external contract of a component (as defined by its interfaces) to the internal realisation of that behaviour by the component's parts; it simply represents the forwarding of signals (operation requests or events). In our case, the delegation connector is used at the composite component level so that its 'parts' become the constituent components.

The UML 2.0 specification restricts the use of the delegation connector to interfaces of the same kind. A further constraint is that the target interface of the connector must support a signature compatible subset of operations of the source interface. In fact, both constraints are formally put in Definition 5.1.3 which gives the construction for the composite component's signature. Recall that the composite signature \( \Sigma_1 \oplus \Sigma_2 \) is formed from the signatures of the constituent components by eliminating all interfaces participating in communication (assembly connector(s) can be used for those). The idea is that we may use delegation connectors for the interfaces of the composite signature.

The semantics of the delegation connector, given in UML 2.0 (pp.160-161 in [OMG04]), is that behaviour which is available on a component may not actually be realised by that component itself, but by another component within it. The essence of this description of the semantics can be formally captured by Definition 5.1.4. In particular, the definition says that behaviour of the composite, on non-connected interfaces, comprises behaviour from each of the constituent components - depending on which component the interface in question belongs to. Therefore, we propose the use of the UML delegation connector for representing the non-connected interfaces of the constituent components at the composite component level.

A delegation connector in UML 2.0 is represented by an open arrow which is stereotyped
by \texttt{\textless \textless delegate\textgreater \textgreater}. The direction of the arrow\(^1\) is from the interface of the composite towards that of the constituent component in the case of provided interfaces (see, for example, interface \texttt{ISearchFre} in Figure 5.1) and from the interface of the constituent component towards that of the composite in the case of a delegation connector between required interfaces (see interface \texttt{IDetectSignal} in Figure 5.1).

## 5.3 Well-behavedness of the Composite Component

Based on Definition 5.1.3 and Definition 5.1.5 we have formally defined a notion of composition of components, which is associative and commutative. Essentially, the signature of the resulting system is defined to be the composite of the components' signatures. The dynamics of the system reflect the fact that a behaviour involves behaviours from each of the components and that these must agree on connected interfaces.

In Section 4.2 we considered constraints on the vector language of a single component that ensure it is well-behaved. Discreteness guarantees that only a finite number of events may occur within finite time, and ensures that there is an initial point in time in which nothing has happened. Local left-closure guarantees that every occurrence of an event (e.g. call to an operation) at an interface of the component is recorded in its component language, i.e. there is a component vector in \( V \) to describe it.

In this section, we concentrate on the effect of composition on well-behaved components. In particular, we are interested in preservation of discreteness and local left-closure in the composite component.

We have seen (Definition 5.1.4) that the notion of consistency among component vectors identifies pairs of vectors (one from each component language) which agree on the coordinate corresponding to the connected interfaces of the two components. We have also seen that discreteness and local left-closure are constraints imposed on the order structure of a component language and thus are also dependent on context - on what other vectors are in the language. Therefore, the notion of consistency among

\(^1\)This is not entirely clear in the UML specification document [OMG04] but can be reasonably derived from its consistent use throughout the document.
pairs of component vectors is not adequate for the treatment of such properties under composition of component languages.

This is because, given two consistent component vectors, there is no way to guarantee that all other vectors which describe earlier behaviour than these vectors, in the respective components, also have a consistent counterpart in the other component language (i.e. can be matched under consistency).

It transpires that we need a notion of step by step consistency across the component languages involved. This leads to the notion of compatibility between components which is defined next. It provides a somewhat stronger condition than consistency among pairs of component vectors, effectively lifting the notion to the language level, and incorporates the usual consistency among signatures.

**Definition 5.3.1.** Suppose that $c_1 = (\Sigma_1, V_1)$ and $c_2 = (\Sigma_2, V_2)$ are components. Then, they are compatible if and only if

1. $c_1$ and $c_2$ are consistent

2. If $y_1 \in V_1$ and $y_2 \in V_2$ such that $y_1 \downarrow y_2$ then

   (a) If $u_1 \in V_1$ such that $u_1 \leq y_1$ then $\exists u_2 \in V_2$ such that $y_2 \leq u_2$ and $u_1 \downarrow u_2$

   (b) If $u_2 \in V_2$ such that $u_2 \leq y_2$ then $\exists u_1 \in V_1$ such that $u_1 \leq y_1$ and $u_1 \downarrow u_2$

   (c) If $u \in V_1 \oplus V_2$ and $u \leq y_1 \oplus y_2$ then $\exists u_1 \in V_1$ and $u_2 \in V_2$ such that $u_1 \downarrow u_2$, $u_1 \leq u_1$, $u_2 \leq u_2$, and $u = u_1 \oplus u_2$

Note that the definition of compatibility given in [MS04b] includes an additional condition relating pairwise consistent component vectors and their least upper bounds. Subsequent analysis has shown that this can be included in a more general treatment of the interrelation between the composition operators $\downarrow$, $\oplus$ and least upper and greatest lower bounds (cf Lemma 5.3.2).

Now we are set to consider whether the composite component obtained by combining compatible components, following the construction given in Definition 5.1.5, is well-behaved. It can be readily shown that the composite $c_1 \oplus c_2$ is locally left-closed whenever $c_1$ and $c_2$ are locally left-closed and compatible components.
5.3. Well-behavedness of the Composite Component

**Lemma 5.3.1.** If \( c_1 \) and \( c_2 \) are compatible components which are locally left-closed, then \( c_1 \oplus c_2 \) is locally left-closed.

**Proof.**

Let \( y \in V_1 \oplus V_2 \) and let \( i \in I_{\Sigma_1} \Delta I_{\Sigma_2} \) and let \( A < x < y(i) \), then \( y = y_1 \oplus y_2 \) for \( y_1 \in V_1 \) and \( y_2 \in V_2 \). Without loss of generality let \( i \in I_{\Sigma_1} \setminus I_{\Sigma_2} \) so that \( y(i) = y_1(i) \).

By local left-closure of \( c_1 \), there exists \( u_1 \in V_1 \) such that \( u_1 \preceq y_1 \) and \( y_1(i) = x \).

By Definition 5.3.1(2b), there exists \( u_2 \in V_2 \) such that \( u_2 \preceq y_2 \) and \( u_1 \preceq u_2 \).

Thus \( u_1 \oplus u_2 \leq u_1 \oplus y_2 = v \), by Remark 5.1.1. So we have \( u_1 \oplus u_2 \in V_1 \oplus V_2 \) and \( u_1 \oplus u_2 \leq v \) and \( (u_1 \oplus u_2)(i) = x \) (since \( i \in I_{\Sigma_1} \setminus I_{\Sigma_2} \)) which means precisely that \( c_1 \oplus c_2 \) is locally left-closed. \( \square \)

In order to prove that the composite component is well-behaved, we must further show that it is discrete. The proof of discreteness for the composite component language is more involved and we introduce it in two steps (Lemma 5.3.2 and Lemma 5.3.3). The following lemma describes the interaction between \( \downarrow \), \( \oplus \) and least upper and greatest lower bounds in a component language. We state the lemma for least upper bounds, for readability, but the result also applies to greatest lower bounds.

**Lemma 5.3.2.** Suppose that \( \Sigma_1, \Sigma_2 \) are consistent signatures and that \( y_1, u_1 \in V_1 \) and \( y_2, u_2 \in V_2 \) such that

1. \( u_1 \downarrow u_2 \) and \( u_1 \downarrow v_2 \)
2. \( y_1 \uplus y_2 \in V_1 \) and \( u_2 \uplus v_2 \in V_2 \)

then,

1. \( (u_1 \uplus y_1) \downarrow (u_2 \uplus v_2) \)
2. \( (y_1 \uplus u_2) \uplus (y_2 \uplus u_2) \in V_1 \oplus V_2 \)
3. \( (y_1 \uplus u_2) \uplus (y_2 \uplus u_2) = (y_1 \uplus y_1) \oplus (u_2 \uplus v_2) \)

**Proof.**
Let $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$. Since $u_1 \downarrow u_2$ and $v_1 \downarrow v_2$, we have $u_1(i) = u_2(i)$ and $v_1(i) = v_2(i)$, and so

\[(u_1 \sqcup u_1)(i) = \max(u_1(i), v_1(i)) \]
\[= \max(u_2(i), v_2(i)) \]
\[= (u_2 \sqcup v_2)(i) \]

Thus, $(u_1 \sqcup u_1) \downarrow (u_2 \sqcup v_2)$, establishing (1).

Suppose, next, that $i \in I_{\Sigma_1} \setminus I_{\Sigma_2}$. Then, $(u_1 \oplus u_2)(i) = u_1(i)$ and $(v_1 \oplus v_2)(i) = v_1(i)$. Now, $u_1 \sqcup v_1 \in V_1$ which implies that $\max(u_1(i), v_1(i))$ is defined and hence, $\max((u_1 \oplus u_2)(i), (v_1 \oplus v_2)(i))$ is defined. This also holds when $i \in I_{\Sigma_2} \setminus I_{\Sigma_1}$, by symmetry. It follows that $(u_1 \oplus u_2) \sqcup (v_1 \oplus v_2) \in V_1 \oplus V_2$, establishing (2).

Finally, if $i \in I_{\Sigma_1} \setminus I_{\Sigma_2}$, then

\[((u_1 \oplus u_2) \sqcup (v_1 \oplus v_2))(i) = \max((u_1 \oplus u_2)(i), (v_1 \oplus v_2)(i)) \]
\[= \max(u_1(i), v_1(i)) \]
\[= (u_1 \sqcup v_1)(i) \]
\[= ((u_1 \sqcup v_1) \oplus (u_2 \sqcup v_2))(i) \]

and similarly when $i \in I_{\Sigma_2} \setminus I_{\Sigma_1}$. It follows that

\[(u_1 \oplus u_2) \sqcup (v_1 \oplus v_2) = (u_1 \sqcup v_1) \oplus (u_2 \sqcup v_2) \]

establishing (3), which completes the proof. □

The proof is similar for the interaction between $\sqcap$, $\oplus$ and greatest lower bounds (replace $\sqcap$ by $\sqcup$ and $\max$ by $\min$).

The lemma postulates that the least upper bounds of component vectors which have a consistent counterpart in the other component's language, are themselves consistent (point of (1) of the lemma) and that the least upper bound of the composite component vectors is defined in the composite component language ((2) of the lemma) and is equal to the composite of the least upper bound of the component vectors from each individual component language ((3) of the lemma).
5.3. Well-behavedness of the Composite Component

The following lemma relates to discreteness of the composite component. It establishes that whenever two component vectors in a composite component language describe an earlier part of behaviour than a third vector, also in the language, then their least upper bound and greatest lower bound are defined and belong to the composite language.

**Lemma 5.3.3.** Suppose that \( c_1 = (\Sigma_1, V_1) \) and \( c_2 = (\Sigma_2, V_2) \) are compatible well-behaved components and \( u, v, w \in V_1 \oplus V_2 \) such that \( u, v \leq w \), then

1. \( u \cup v \in V_1 \oplus V_2 \)
2. \( u \cap v \in V_1 \oplus V_2 \)

**Proof.**

We begin by proving point (1) of the lemma. Since \( u \in V_1 \oplus V_2 \), by definition (Definition 5.1.5), there exist \( w_1 \in V_1 \) and \( w_2 \in V_2 \) such that \( u = w_1 \oplus w_2 \). Since \( u \leq w = u_1 \oplus u_2 \), by 2(c) of Definition 5.3.1 there exist \( u_1 \in V_1 \) and \( u_2 \in V_2 \) such that \( u_1 \downarrow u_2 \) and \( u_1 \leq w_1, u_2 \leq w_2 \). Similarly, there exist \( v_1 \in V_1 \) and \( v_2 \in V_2 \) such that \( v_1 \downarrow v_2 \) and \( v_1 \leq w_1, v_2 \leq w_2 \). Since \( u, v \leq w \), and \( w \in V_1 \) and \( c \) is well-behaved (and thus, also discrete), we can deduce that \( u \cup v \in V_1 \), by Definition 4.2.2. Similarly, we can deduce that \( u \cap v \in V_2 \).

So we have shown that \( u_1 \downarrow u_2, v_1 \downarrow v_2 \) and \( u_1 \cup v_1 \in V_1, u_2 \cup v_2 \in V_2 \). Thus, by (2) of Lemma 5.3.2 we can deduce that \( (u_1 \oplus u_2) \cup (v_1 \oplus v_2) \in V_1 \oplus V_2 \), hence, \( u \cup v \in V_1 \oplus V_2 \), establishing (1).

Next, we prove point (2) of the lemma. Since \( u, v \in V_1 \oplus V_2 \), by definition there exist \( u_1, u_2 \in V_1 \) and \( u_2, v_2 \in V_2 \) such that \( u = u_1 \oplus u_2 \) and \( v = v_1 \oplus v_2 \). As in the proof of point (1) above, by well-behavedness of \( c_1, c_2 \) we may deduce that \( u_1 \cap u_2 \in V_1 \) and \( v_1 \cap v_2 \in V_2 \).

Since \( u_1 \oplus u_2 \) and \( v_1 \oplus v_2 \) both exist, we must have \( u_1(i) = u_2(i) \) and \( v_1(i) = v_2(i) \), for each \( i \in \Sigma_1 \cap \Sigma_2 \). Thus, \( \min(u_1(i), v_1(i)) = \min(u_2(i), v_2(i)) \), for each \( i \in \Sigma_1 \cap \Sigma_2 \), which means that \( (u_1 \cap v_1)(i) = (u_2 \cap v_2)(i) \), for each \( i \in \Sigma_1 \cap \Sigma_2 \). By Definition 5.1.4 we may deduce that \( (u_1 \cap v_1) \downarrow (u_2 \cap v_2) \) and hence, \( (u_1 \cap v_1) \oplus (u_2 \cap v_2) \) is defined and
belongs to $V_1 \oplus V_2$. Now by (3) of Lemma 5.3.2, we have

$$(u_1 \oplus u_2) \oplus (u_2 \oplus u_3) = (u_1 \oplus u_2) \oplus (u_2 \oplus u_2)$$

Thus, $(u_1 \oplus u_2) \oplus (u_2 \oplus u_2) \in V_1 \oplus V_2$ and hence $u_1 \oplus u_2 \in V_1 \oplus V_2$ establishing (2). □

The above results are summarised in the following theorem which is the main result of this section.

**Theorem 5.3.1.** If $c_1$ and $c_2$ are compatible well-behaved components, then $c_1 \oplus c_2$ is a well-behaved component.

**Proof.**

Since $c_1$ is a well-behaved component, we have $A_{c_1} \in V_1$. Similarly, we have $A_{c_2} \in V_2$.

Thus, $A_{c_1 \oplus c_2} \in V_1 \oplus V_2$. Now by Lemma 5.3.3, we may deduce that $c_1 \oplus c_2$ is discrete. By Lemma 5.3.1 it is locally left-closed. Thus, $c_1 \oplus c_2$ is well-behaved. □

Therefore, we have shown that under certain conditions, captured by the notion of compatible components, two well-behaved components can be put together and the resulting composite shall also be well-behaved.

In this way, we can establish well-behavedness at the individual component level and check for compatibility. If this turns out to be the case, then the composition of the components results in a composite component which is guaranteed to be well-behaved. Considering that composition in our approach is associative, this allows for building well-behaved systems out of well-behaved components.

### 5.4 Illustration by example

We illustrate the formal notion of composition in our approach by means of an extended example. In particular, we apply the formalism introduced in this chapter to describe the composition of the CMenu and CTuner components of our case study considered in Section 4.4.

We start by describing the role of CTuner in the context of the overall component specification architecture of Figure 4.5. The CTuner component is concerned with the
5.4. Illustration by example

The task of tuning a TV set to a given frequency. When the frequency of the tuner is changed, it produces noise which might result in undesired artifacts on the TV screen and speakers. Therefore, it is a product requirement that the screen and the speakers should be blanked (blocked) before the frequency is changed. As soon as the tuner is tuned to the new frequency, the screen and the speakers can be unblanked. More details on the functionality of the tuner can be found in [vO03]. Figure 5.2 shows the configuration of components required for this functionality and includes the names for the interfaces between participating components. Notice that it is part of the overall configuration of Figure 4.5.

![Diagram of component specification architecture for the tuning task](image)

Figure 5.2: Component specification architecture for the tuning task

We give a formal description of the CTuner component by applying Definition 3.1.3, as done in Section 4.4 for the CMenu component.

The CTuner component has one provided interface, IDetectSignal, and three required interfaces, ISpeaker, IScreen, and IAntenna. Hence, the provided interfaces of CTuner are given by the set \( P_T = \{ \text{IDetectSignal} \} \) and the required interfaces are given by the set \( R_T = \{ \text{ISpeaker, IScreen, IAntenna} \} \). Thus, the set of interfaces is given by \( I_T = \{ \text{IDetectSignal, ISpeaker, IScreen, IAntenna} \} \). We check that \( P_T \cap R_T = \emptyset \).

The function \( \beta \) as defined in Definition 3.1.1 specifies the set of operations associated with each interface. In the case of CTuner, we have,
\[\beta_T(IDetectSignal) = \{c1\}\]
\[\beta_T(ISpeaker) = \{d1, d2\}\]
\[\beta_T(IScreen) = \{f1, f2\}\]
\[\beta_T(IAntenna) = \{g1\}\]

where \(c1\) is an abbreviation for operation \(detectSignal()\), as in Section 4.4, while \(d1, f1\) are abbreviations for operation \(drop()\) on ISpeaker and IScreen respectively, \(d2, f2\) for \(restore()\) on ISpeaker and IScreen respectively, and \(g1\) for \(tune()\). Recall that operation names are unique on interfaces in our formal framework.

Suppose that a component developer considers the intended behaviour of CTuner fulfilling the following:

- Once the CMenu component requests a change of frequency (which corresponds to invoking operation \(c1\) on IDetectSignal) the CTuner component immediately calls both CAudio and CVideo components to blank the screen and speakers, by dropping their signal.

- Once both screen and speakers have been blanked, the CTuner component may proceed to change the frequency on the CTunerDriver component.

- Once the new signal has been detected and the CTuner is tuned to the new frequency, and only then, the CTuner immediately calls both CAudio and CVideo to unblank the screen and speakers at the same time, by restoring their signal.

Notice that the requests to CAudio and CVideo for restoring their signal are intended to occur at the same time so that picture and sound are restored at once.

This informal description of behaviour of the CTuner component is captured in the UML sequence diagram of Figure 5.3 which describes the allowed sequences of events on its interfaces. Notice that the mode of communication within a TV platform is synchronous and this is represented by the use of filled arrowheads in the diagram. By unfolding the sequence diagram into component vectors, following the formal construction described in Section 3.2, we obtain the set of vectors that indicate the intended
5.4. Illustration by example

Figure 5.3: The CTuner component performing the tuning task

behaviour of the CTuner component and comprise its corresponding component language. To avoid overelaborating the example, we do not give the details of the unfolding here. It can be easily checked against the material and examples of Section 3.2.

If we write \((z, w, x, y)\) for the functions \(v\) of Definition 3.1.2 with \(v(I\text{DetectSignal}) = z, \ v(I\text{Speaker}) = w, \ v(I\text{Screen}) = x\) and \(v(I\text{Antenna}) = y\) we obtain the following set of component vectors for CTuner.

\[
V_T = \{ (A, A, A, A), (c1, A, A, A), (c1, d1, A, A), (c1, A, f1, A), \\
(c1, d1, f1, A), (c1, d1, f1, g1), (c1, d1d2, f1f2, g1) \}
\]

The component language \(V_T\) is discrete and locally left-closed, so CTuner is a well-behaved component. We do not give the proof here. This is left to the interested reader. (It follows from similar reasoning to that applied to \(V_M\) of the CMenu component in Section 4.4.)

We may now consider the composition of the CTuner and CMenu components. Let \(c_M = (\Sigma_M, V_M)\) denote the CMenu component and \(c_T = (\Sigma_T, V_T)\) denote the CTuner component. The composition of \(c_M, c_T\) involves composing their signatures and their
component languages. We start by considering their signatures. First, we check whether they are consistent against Definition 5.1.1.

The two components must have no provided and no required interfaces in common. Indeed, \( P_M \cap P_T = \emptyset \) and \( R_M \cap R_T = \emptyset \). They do have an interface in common, that is, IDetectSignal which is a required interface of CMenu and a provided interface of CTuner. Hence, we have that \( I_{\Sigma_M} \cap I_{\Sigma_T} = \{ \text{IDetectSignal} \} \) for which we also have that \( \beta_M(\text{IDetectSignal}) = \{ \text{c1} \} = \beta_T(\text{IDetectSignal}) \). Thus, in the notation of Definition 5.1.1 we have \( \Sigma_M \downarrow \Sigma_T \) and the two signatures are consistent. Consequently, composition can potentially take place over IDetectSignal (see also Figure 5.1). Having established that \( \Sigma_M \downarrow \Sigma_T \) we may now obtain the composite signature \( \Sigma_{MT} = \Sigma_M \oplus \Sigma_T \) following the construction given in Definition 5.1.3.

We have \( P_{MT} = (P_M \cup P_T) \setminus (R_M \cup R_T) = \{ \text{ISelect, IItemSelection} \} \). Note that IDetectSignal does not appear in \( P_{MT} \), though it is in \( P_T \), because it also belongs to \( R_M \).

Also, \( R_{MT} = (R_M \cup R_T) \setminus (P_M \cup P_T) = \{ \text{ISpeaker, IScreen, IAntenna} \} \). Note that IDetectSignal does not appear in \( R_{MT} \) because it belongs to \( P_T \).

The function \( \beta_{MT} \) on the interfaces of the composite component \( c_M \oplus c_T \) satisfies \( \beta_{MT}(i) = \beta_k(i) \) whenever \( i \in I_{\Sigma_k}, k = M, T \). For instance, in the case of IFineTune, we have \( \beta_{MT}(\text{IFineTune}) = \beta_M(\text{IFineTune}) = \{ b1, b2, b3 \} \) since \( \text{IFineTune} \in I_{\Sigma_M} \).

Up to this point we have checked for consistency among the signatures of the two components and subsequently defined the signature of the composite component \( c_M \oplus c_T \).

Now we turn our attention to composing the component vectors from each component. Definition 5.1.5 dictates that a composite component vector comprises component vectors from each language so long as these agree on the coordinates corresponding to the connected interfaces. The latter requirement is imposed by consistency among vectors as given in Definition 5.1.4.

For the component languages of CMenu and CTuner this amounts to checking whether the sequence appearing on the third (last) coordinate of the component vectors in \( V_M \) is the same as that of the first coordinate of component vectors in \( V_T \). In what follows, we
describe incrementally how the component vectors of the composite language \( V_M \oplus V_T \) are obtained.

The CMenu component engages in a number of events before making a call to operation \( c1 \) on CTuner. The resulting behaviours are described by the following set of component vectors, in which \( c1 \) has not occurred yet.

\[
(A, A, A), (a1, A, A), (a1, b1, A), (a1a2, A, A), (a1, b1b2, A), (a1, b1b3, A), (a1a2, b1, A)
\]

These vectors are consistent with \((A, A, A, A)\) from \( V_T \) of CTuner. Their composition with this vector yields the following set of composite component vectors. For example,

\[
(a1, A, A) \oplus (A, A, A, A) = (a1, A, A, A, A).
\]

\[
X = \{(A, A, A, A, A), (a1, A, A, A, A), (a1, b1, A, A, A), (a1a2, A, A, A, A), (a1, b1b2, A, A, A), (a1, b1b3, A, A, A), (a1a2, b1, A, A, A)\}
\]

This set reflects the fact that the CTuner component does nothing while CMenu has not issued \( c1 \). The occurrence of \( c1 \) on IDetectorSignal is recorded in the component vector \((a1a2, A, c1) \in V_M\) which is the smallest vector in \( V_M \) that has \( c1 \) on its third coordinate. This occurrence is recorded in \( V_T \) by \((c1, A, A, A)\) which is also the smallest vector in \( V_M \) which contains \( c1 \). After this occurrence, CTuner engages in the tuning task and performs its part, by blanking CAudio, CVid, tuning the tuner driver and then unblanking CAudio and CVid. This behaviour is captured by the following set of component vectors in \( V_T \).

\[
(c1, A, A, A), (c1, d1, A, A), (c1, A, f1, A), (c1, d1, f1, A), (c1, d1, f1, g1), (c1, d1d2, f1f2, g1)
\]

These vectors are consistent with \((a1a2, A, c1)\) from \( V_M \) of CMenu. Their composition with this vector yields the following set of composite component vectors. For example,

\[
(a1a2, A, c1) \oplus (a1, d1, A, A) = (a1a2, A, d1, A, A).
\]

\[
Y = \{(a1a2, A, A, A, A), (a1a2, A, d1, A, A), (a1a2, A, A, f1, A), (a1a2, A, d1, f1, A), (a1a2, A, d1, f1, g1), (a1a2, A, d1d2, f1f2, g1)\}
\]

This set reflects the fact that the CMenu component does nothing while CTuner manages its downstream devices (CAudio and CVid) in fulfilling the \( c1 \) request. The occurrences of \( d2 \) and \( f2 \) are recorded in the component vector \((c1, d1d2, f1f2, g1) \in V_T\).
and signify completion of the request, since only then are the speakers and screen set to the new frequency.

This allows the CMenu component to engage in its subsequent behaviour described by the following set of component vectors in $V_M$.

$$(a1a2, b1, c1), (a1a2, b1b2, c1), (a1a2, b1b3, c1)$$

These vectors are consistent with vector $(c1, d1d2, f1f2, g1)$ from $V_T$ of CTuner. Their composition with this vector yields the following set of composite component vectors. For example, $(a1a2, b1, c1) \oplus (c1, d1d2, f1f2, g1) = (a1a2, b1, d1d2, f1f2, g1)$.

$$Z = \{(a1a2, b1, d1d2, f1f2, g1), (a1a2, b1b2, d1d2, f1f2, g1), (a1a2, b1b3, d1d2, f1f2, g1)\}$$

From this point onwards the behaviour of the two components with regard to the tuning task has been completed and can only be repeated as a whole. Thus, the resulting composite component language is given by the union of the sets $X, Y, Z$ as,

$$V_M \oplus V_T = \{(a, a, a, a, a), (a1, a, a, a, a), (a1, b1, a, a, a), (a1a2, a, a, a, a),$$

$$(a1, b1b2, a, a, a), (a1, b1b3, a, a, a), (a1a2, b1, a, a, a),$$

$$(a1a2, d1, a, a, a), (a1a2, d1d2, a, a, a), (a1a2, d1d2, f1f2, g1),$$

$$(a1a2, d1d2, f1f2, g1), (a1a2, b1b2, d1d2, f1f2, g1), \}$$

In order to determine whether the resulting composite component $c_M \oplus c_T$ is well-behaved, it suffices to check whether the components $c_M$ and $c_T$ are compatible in the sense of Definition 5.3.1. If this is the case, then Theorem 5.3.1 guarantees that $c_M \oplus c_T$ is well-behaved.

We have already seen that $\Sigma_M \downarrow \Sigma_T$ and thus $c_M \downarrow c_T$ which means that (1) of Definition 5.3.1 is satisfied. We now check (2) of the definition. This involves a check between all pairs of consistent vectors considered in obtaining $V_M \oplus V_T$. We give three examples here, one from each subset $X, Y, Z$ that comprise $V_M \oplus V_T$. The rest can be checked following similar reasoning.
Consider the case of \( \mathbf{u}_1 = (a_1, A, A) \in V_M \) and \( \mathbf{v}_2 = (A, A, A, A) \in V_T \) for which \((a_1, A, A) \downarrow (A, A, A, A)\).

We have \( \mathbf{u}_1 = (A, A, A) \in V_M \) such that \( \mathbf{u}_1 \leq \mathbf{u}_1 \). There exists \( \mathbf{u}_2 = (A, A, A, A) \in V_T \) such that \( \mathbf{u}_2 \leq \mathbf{v}_2 \) and \( \mathbf{u}_1 \downarrow \mathbf{u}_2 \). Thus, condition 2(a) of the definition holds.

We also have that \( \mathbf{u}_2 = (A, A, A, A) \in V_M \) such that \( \mathbf{u}_2 \leq \mathbf{u}_2 \). There exists \( \mathbf{u}_1 = (A, A, A) \in V_T \) such that \( \mathbf{u}_1 \leq \mathbf{u}_1 \) and \( \mathbf{u}_1 \downarrow \mathbf{u}_2 \). Thus, condition 2(b) of the definition holds.

Finally, let \( \mathbf{u} = (A, A, A, A) \in V_M \) such that \( \mathbf{u} \leq \mathbf{u} \). There exists \( \mathbf{u}_2 = (a_1, A, A, A) \in V_T \) such that \( \mathbf{u}_2 \leq \mathbf{u}_2 \) and \( \mathbf{u} = \mathbf{u}_1 \oplus \mathbf{u}_2 \). Thus, condition 2(c) of the definition holds.

Next consider a case from the set \( Y \). Let vector \( \mathbf{v}_1 = (a_1, A, c_1) \in V_M \) and let \( \mathbf{v}_2 = (a_1, d_1, A, A) \in V_T \) for which \( \mathbf{v}_1 \downarrow \mathbf{v}_2 \).

We have \( \mathbf{u}_1 = (a_1, A, A, A) \in V_M \) such that \( \mathbf{u}_1 \leq \mathbf{u}_1 \). There exists \( \mathbf{u}_2 = (A, A, A, A) \in V_T \) such that \( \mathbf{u}_2 \leq \mathbf{v}_2 \) and \( \mathbf{u}_1 \downarrow \mathbf{u}_2 \). Thus, condition 2(a) of the definition holds.

We also have that \( \mathbf{u}_2 = (A, A, A, A) \in V_M \) such that \( \mathbf{u}_2 \leq \mathbf{u}_2 \). There exists \( \mathbf{u}_1 = (a_1, A, A) \in V_T \) such that \( \mathbf{u}_1 \leq \mathbf{u}_1 \) and \( \mathbf{u}_1 \downarrow \mathbf{u}_2 \). Thus, condition 2(b) of the definition holds.

Finally, let \( \mathbf{u} = (a_1, A, A, A) \in V_M \) such that \( \mathbf{u} \leq \mathbf{u} \). There exists \( \mathbf{u}_2 = (a_1, A, A, A) \in V_T \) such that \( \mathbf{u}_2 \leq \mathbf{u}_2 \) and \( \mathbf{u} = \mathbf{u}_1 \oplus \mathbf{u}_2 \). Thus, condition 2(c) of the definition holds.

Next consider a case from the set \( Z \). Let vector \( \mathbf{v}_1 = (a_1, b_1, c_1) \in V_M \) and \( \mathbf{v}_2 = (a_1, d_1, a_1, A, A) \in V_T \) for which \( \mathbf{v}_1 \downarrow \mathbf{v}_2 \).

We have \( \mathbf{u}_1 = (a_1, A, A, A) \in V_M \) such that \( \mathbf{u}_1 \leq \mathbf{u}_1 \). There exists \( \mathbf{u}_2 = (a_1, A, A, A) \in V_T \) such that \( \mathbf{u}_2 \leq \mathbf{u}_2 \) and \( \mathbf{u}_1 \downarrow \mathbf{u}_2 \). Thus, condition 2(a) of the definition holds.

We also have that \( \mathbf{u}_2 = (a_1, A, A, A) \in V_M \) such that \( \mathbf{u}_2 \leq \mathbf{u}_2 \). There exists \( \mathbf{u}_1 = (a_1, A, A) \in V_T \) such that \( \mathbf{u}_1 \leq \mathbf{u}_1 \) and \( \mathbf{u}_1 \downarrow \mathbf{u}_2 \). Thus, condition 2(b) of the definition holds.

Finally, let \( \mathbf{u} = (a_1, A, d_1, f_1, g_1) \in V_M \) such that \( \mathbf{u} \leq \mathbf{u} \). For this vector we have \( \mathbf{u} \leq \mathbf{v} = (a_1, A, A, A, A) \in V_T \) such that \( \mathbf{u} \leq \mathbf{u} \) and \( \mathbf{u} = (a_1, A, A, A, A) \in V_T \).
(c1, d1, f1, g1) ∈ V_T such that u_1 ⊑ u_2, u_1 ≤ u_2, u_2 ≤ u_3 and u = u_1 ⊕ u_2. Thus, condition 2(c) of the definition holds.

Using similar reasoning for the remaining vectors related by ⊑, we conclude by Definition 5.3.1 that c_M, c_T are compatible components. We also have that c_M and c_T are well-behaved. Thus, by Theorem 5.3.1, c_M ⊕ c_T is well-behaved.

5.5 Concluding note

In this chapter, we presented the notion of composition of components considered in our formal framework. Components are composed by connecting a required interface of one component to a provided interface of the other. The notion of consistency between signatures serves to identify interfaces that are eligible for connection. We also allowed for some flexibility on the strict interface typing aspect of consistent signatures (Lemma 5.1.1). The notion of consistency among component vectors from each language determines which pairs of component vectors are eligible for composition. The composite component language brings together all pairs of vectors that have been matched for consistency.

In our case study of Section 5.4 it became apparent that not all such pairs result (under composition ⊕) in component vectors that describe intended behaviour only of the composite. It appears that in deciding which component vectors (out of the consistent ones) should be composed and thus be part of the resulting composite language, we need to take into account the order structure of the individual component languages. At the moment, this is only done for component vectors from each language that adhere to Remark 5.1.1, but this does not cover any pair of consistent vectors. We will have more to say about this in the concluding chapter of the thesis.

It was also shown that the operation of composition on components is associative and commutative. Thus, the resulting composite can be further composed with other components and composites to form a larger system.

UML structure diagrams were proposed for visualising connected components, providing these are eligible for composition. This involves a more concrete semantics for the
assembly connector (in terms of consistency among component vectors) and the delegation connector (in terms of consistency among signatures) used in composite structure diagrams.

In Chapter 4 we required certain properties of component languages that lead to the characterisation of well-behaved components. With the goal of predictable assembly in mind, we examined the preservation of well-behavedness under composition. It turned out that this is the case so long as the components in question are compatible. The compatibility condition effectively requires a step by step consistency among component vectors, in the sense that if two vectors are consistent, then all vectors describing earlier behaviour in the respective languages also have a consistent counterpart.

The idea behind well-behavedness is that in reasoning about these component properties we may identify potentially pathological behaviour. Such behaviour may be the result of inconsistencies in sequence diagrams such as a race condition. The preservation of well-behavedness under composition offers interesting perspectives with regard to combining sequence diagrams (scenarios). Given that the CMem and CTuner components in our case study are well-behaved and compatible, the combination of the respective sequence diagrams (Figure 4.8, Figure 5.3) would yield a sequence diagram in which no valid (implicit) scenarios, other than what is explicitly described in the diagram, can appear along the lifelines of the two components.
Chapter 6

Event-oriented Description of Component Behaviour

The behaviour of component-based systems often involves a complex interplay of action/reaction relationships varying over time. We have argued that a suitable notion of behaviour in this context is one that expresses the patterns of actions a component can perform.

In this chapter, we associate well-behaved components with behavioural presentations which comprise a behavioural model that focuses on describing occurrences of events over time. This provides an event-oriented description of component behaviour, which is particularly useful for replacement and adaptation in a component setting.

Behavioural presentations are order theoretic structures, and establishing their relationship to component languages effectively builds a bridge between algebraic and order theoretic representation of component behaviour. In this way, our formal framework can be related to a more general theory of non-interleaving representation of behaviour [Shi97].

One of the benefits of using the behavioural presentation model for the description of component behaviour is that various temporal relations can be derived from this model in such a way that nondeterminism, concurrency and simultaneity can be treated as distinct phenomena. The temporal relations are used to describe the reactive behaviour of
a component in terms of the time ordering between calls to operations on its interfaces. Whenever this time ordering is respected, we may faithfully expect the component to behave in predictable ways.

The relationship between behavioural presentations and component languages builds on a key technical result due to M. W. Shields, that first appeared in [Shi92], and underlies the relationship between order theoretic objects and language theoretic objects. The behavioural presentation model [Shi88] has been extensively studied in relation to vector languages in [Shi97].

In [MSKF03] we have been concerned with an adaptation of this work for component languages. This involved the characterisation of primes in component languages and a construction for extracting from a component language a quadruple that mirrors a behavioural presentation. [MKS05] is an elaboration of this work that stresses the relationship between discrete and locally left-closed component languages and discrete behavioural presentations. The formal construction was applied to a case study and we have shown how the resulting behavioural presentation can be used to model the relationship between any pair of events occurring on component interfaces.

Here, we give an outline description of the above and suggest a reworking of the construction that maps a component language onto a quadruple which mirrors a left-closed behavioural presentation. This has to do with a minor difference in defining the occurrence function (Definition 6.3.2) and providing the more accessible notion of primal vectors (Definition 6.3.1) in component languages. These vectors are then shown to be the primes in a component language, based on the analysis in [SM04b] which contains the complete proofs.

We start by outlining the behavioural presentation model and discuss a particular subclass of behavioural presentations that describe discrete behaviour. Then, we give the formal construction that translates a component language into a behavioural presentation. We illustrate the construction and the resulting event-oriented description of a component by means of an extended example.


6.1 Behavioural Presentations

In this section we give an overview of behavioural presentations, which were introduced in [Shi88], focusing on the temporal relations they determine among occurrences of events.

Any computer system is associated with a set of events. When an event actually happens we talk about an occurrence of that event. Therefore, for any system there is a corresponding set \( E \) of events and a set \( O \) of occurrences of those events. For instance, making a call to operation \( \text{startSearch}() \) on interface \( \text{ISearch} \) of the \( \text{CMenu} \) component in our examples is considered an occurrence of the event \('\text{startSearch}()\) operation call'. The occurrence of a call to operation \( \text{startSearch}() \) causes the \( \text{CMenu} \) component to perform some processing perhaps and when it is ready, in the right state, make an operation call to another component (another occurrence of an event) and so on.

The behavioural presentation model builds on the idea that a description of the possible behaviour of a system may consist of a set of assertions concerning what events have occurred during its execution and the relations between them. An assertion will be valid relative to some point in the space-time of the system. Therefore, each system is associated with a set of points. A point can be thought of as a "possible world" in which certain events have occurred. Each point is identified with the set of occurrences of events which have taken place prior to that point. The intuition is that each point represents that point in time reached after all occurrences which constitute it have taken place.

Events may have multiple occurrences. Two occurrences of the same event are the same if they have been preceded by the same sequences of events. Consider the sequences of events \( aabab \) and \( aaabab \). The second \( b \) in the sequence \( aabab \) is not the same occurrence as the second \( b \) in \( aaabab \). They take place in different "possible worlds". We may thus refer to events by giving the sequence of which they are the last occurrence.

These concepts underlie the behavioural presentation model which will be used for an event-oriented description of component behaviour in our approach. A behavioural
Chapter 6. Event-oriented Description of Component Behaviour

presentation is defined as follows.

**Definition 6.1.1.** A behavioural presentation is a quadruple $B = (O, \Pi, E, \lambda)$, where

1. $O$ is a set of occurrences
2. $\Pi \subseteq \wp(O)$ is a non-empty set of points
3. $E$ is a set of events
4. $\lambda : O \rightarrow E$ is the occurrence function

which satisfies $\bigcup_{\pi \in \Pi} \pi = O$.

The requirement that $\bigcup_{\pi \in \Pi} \pi = O$ says that every occurrence belongs to some point and essentially reflects the fact that we should not be concerned with things that could never happen. The function $\lambda$ associates occurrences with events. Therefore, $\lambda(o) = e$ is to be read as 'o is an occurrence of e'.

In order to obtain a precise description of the dynamic characteristics of the component in hand, we need to model: i) the order in which the component makes calls to operations on other components through its required interfaces, and ii) the order in which the component receives calls to operations from other components on its provided interfaces. The set of occurrences $O$ in a behavioural presentation determines various temporal relations, which can be used to this end.

**Definition 6.1.2.** Let $B$ be a behavioural presentation and suppose that $o_1, o_2 \in O$.

Define,

- $o_1 || o_2 \iff \forall \pi \in \Pi : o_2 \in \pi \Rightarrow o_1 \not\in \pi$ and we say $o_1, o_2$ are mutually exclusive
- $o_1 \rightarrow o_2 \iff \forall \pi \in \Pi : o_2 \in \pi \Rightarrow o_1 \in \pi$ and we say $o_1$ has happened no later than $o_2$
- $o_1 \equiv o_2 \iff (o_1 \rightarrow o_2) \land (o_2 \rightarrow o_1)$ and we say $o_1, o_2$ occurred simultaneously
- $o_1 \bowtie o_2 \iff \neg(o_1 || o_2) \land (o_1 \not\rightarrow o_2) \land (o_2 \not\rightarrow o_1)$ and we say $o_1, o_2$ occurred concurrently
6.1. Behavioural Presentations

- \( o_1 < o_2 \iff (o_1 \rightarrow o_2) \land (o_2 \not\rightarrow o_1) \) and we say \( o_1 \) happened strictly before \( o_2 \)

Using the above temporal relations we can determine the causal and temporal ordering amongst calls to operations occurring at the interfaces of a component, and in this way describe its observable behaviour.

It can be seen that the temporal relations derived from behavioural presentations are based on two fundamental relations: \( \parallel \) and \( \rightarrow \). These relations introduce concepts of mutual exclusion and time ordering among events, in a fashion similar to the well-known conflict and causal temporal relations in [Win88], [KFLO00] and elsewhere. The relation \( \rightarrow \) is a pre-order - that is, a transitive, reflexive relation. If \( o_1 \rightarrow o_2 \), then if \( o_2 \) has happened, then so must \( o_1 \). As for \( \parallel \), if \( o_1 \parallel o_2 \), then an occurrence of \( o_2 \), say, means that \( o_1 \) cannot occur, and vice versa. In short, occurrence of one excludes future occurrence of the other. It is this relation that allows us to introduce notions of nondeterminism into the model. In fact, \( \parallel \) is an independence relation - that is, an irreflexive, symmetric relation. Put formally, \( R \) is an independence relation on \( X \) if and only if

- \( \forall x, y \in X : xRy \Rightarrow yRx \) (symmetry)
- \( \forall x, y \in X : xRy \Rightarrow x \neq y \) (irreflexivity)

Finally, \( < \) is a strict pre-order - that is, a transitive and irreflexive relation. The following remark, found in [Shi97], gives the basic properties of all temporal relations derived from behavioural presentations.

Remark 6.1.1. Suppose that \( B \) is behavioural presentation, then

1. \( \rightarrow \) is a pre-order

2. \( \parallel \) is an independence relation, and whenever \( o_1 \rightarrow o_2 \) and \( o'_1 \rightarrow o'_2 \), then \( o_1 \parallel o'_1 \Rightarrow o_2 \parallel o'_2 \)

3. \( \equiv \) is an equivalence relation

4. \( co \) is an irreflexive and symmetric relation
5. $<$ is a strict pre-order

A pre-ordered set is a pair $(X, \leq)$, where $X$ is a set and $\rightarrow$ is a relation on the set which satisfies:

- $\forall x \in X : x \rightarrow x$ (reflexivity)
- $\forall x, y \in X : x \rightarrow y \wedge y \rightarrow z \Rightarrow x \rightarrow z$ (transitivity)

Notice that $\rightarrow$ is not required to be antisymmetric (in which case it would become a partial order). Thus, if $x \rightarrow y$ and $y \rightarrow x$, instead of $x = y$, we get $x \equiv y$ which means that $x$ and $y$ are distinct but stand in an equivalence relation. This allows the formal treatment of simultaneity, on top of concurrency, in behavioural presentations.

In further explanation of the notation, $\equiv$ is the equivalence relation generated by the pre-order $\rightarrow$. Hence, if $o, o_1, o_2 \in O$ and $o_1 \equiv o_2$, then

- $o_1 \rightarrow o \iff o_2 \rightarrow o$
- $o \rightarrow o_1 \iff o \rightarrow o_2$
- $o \parallel o_1 \iff o \parallel o_2$

In other words, two occurrences related by $\equiv$ stand in exactly the same relationship to other occurrences. Further, suppose that $o_1 \equiv o_2$, and assume that we have some means of deriving from the system the exact time $t$ at which $o_1$ occurred. Then, if $o_1$ is in a relationship with the clock occurrence $t$, then $o_2$ must also be in that relation (and vice versa). Thus, $o_2$ must also have occurred at $t$. The interpretation is that $o_1, o_2$ are simultaneous. Notice that this is not the same as saying that $o_1, o_2$ are concurrent. $o_1 \parallel o_2$ says that neither precedes the other and they are not mutually exclusive.

In a certain important sense, simultaneity is a refinement of the notion of concurrency in event structures [NPW81, Win88]. It is obtained by considering a pre-order rather than a partial order as the causal dependency or time ordering relation. This allows the treatment of concurrency in the same way as in event structures. The fact that a
pre-order is not antisymmetric allows to further derive an equivalence relation which can be used to describe events that occur at exactly the same time.

Having simultaneity as well as concurrency, it is possible to be even more precise in describing the relationships between events occurring on component interfaces during execution of the system. Using the temporal relations of Definition 6.1.2 we may capture the relation between all occurrences, as two occurrences are either

1. mutually exclusive
2. ordered in time
3. simultaneous
4. concurrent
5. strictly ordered in time,

and only one of these relations holds for a pair of occurrences [Shi97].

The behavioural presentation model is closely related to the event structures model [NPW81]. In fact, behavioural presentations mildly generalise event structures [NPW81] in allowing time ordering of events, given in Definition 6.1.2, to be a pre-order (a reflexive and transitive relation) rather than a partial order, thereby allowing the representation of simultaneity as well as concurrency.

The connection between behavioural presentations and event structures is further examined in [Shi97] where it is shown that the relationship between the two involves a form of closure, which comes down to a behavioural presentation being left-closed and coherent. Left-closed behavioural presentations are discussed later (cf. Definition 6.2.3). To define coherence we shall need a couple of definitions (see [Shi97]).

Let \((D, \leq)\) be a poset. Define \(x,y \in D\) to be compatible, denoted by \(x \uparrow y\), if and only if the set \(\{x, y\}\) has an upper bound.

A subset \(X \subseteq D\) is pairwise compatible if and only if for all \(x, y \in X\) we have \(x \uparrow y\).

A poset \((D, \leq)\) is coherent if and only if every pairwise compatible subset of \(D\) has a least upper bound.
Hence, from the order theoretic nature of the behavioural presentation model we may infer that \( B \) is coherent when \((\Pi, \leq)\) is coherent. With regard to its relation to the event structures model, it turns out that a prime event structure, as defined in [Win88], corresponds to a left-closed and coherent behavioural presentation for which \( co \neq \emptyset \) and \( \equiv \) is the identity relation in \( O \). Further details can be found in [Shi97].

### 6.2 Discrete Behavioural Presentations

Component-based systems are largely conceived of as proceeding in discrete steps. This implies that occurrences of events in the system do not blur into one another. In the spirit of [Win88], this means that any two events in the system can be separated by an open neighbourhood. The famous Zeno paradoxes\(^1\), in which the philosopher seeks to demonstrate the impossibility of motion, are examples of a non-discrete representation of systems. We shall not be concerned with such descriptions of systems here since events in our approach are considered instantaneous.

The behavioural presentation model can be used to describe systems which proceed in discrete steps. For this purpose, we shall consider a subclass of behavioural presentations, namely discrete behavioural presentations [Shi88, Shi97], which are well suited for representing discrete behaviour. Before defining discrete behavioural presentations we discuss related properties that motivate the definition. We will see that the properties that characterise them are related to the properties of well-behaved components.

First, we want to ensure that discrete systems proceed in an orderly way - in discrete steps. A step in behavioural presentations is considered in the following terms. Assume that the system is in a state where its occurrences of events so far are described by \( \pi \). An occurrence \( o \) of some event takes place and this additional occurrence is now described by \( \pi' \). Thus, we obtain \( \pi' \) by adding in \( o \), to whatever occurrences were

\(^1\) [Shi97] considers Zeno’s arrow paradox and the one that concerns Achilles and the tortoise. The conclusion drawn there from Zeno’s arguments is not that motion is impossible, but that it is not discrete. Note that it is possible to use a behavioural presentation, albeit not a discrete one, to model the situations described in Zeno’s paradoxes, as shown in [Shi97]. In this thesis we shall be concerned with the subclass of discrete behavioural presentations though.
already in \( \pi \). But if \( o' \) is some other occurrence such that \( o \equiv o' \), then \( o' \) must also be in \( \pi' \) (by Definition 6.1.2 of \( \equiv \) and \( \rightarrow \)). Consequently, in describing what happened in moving from \( \pi \) to \( \pi' \) we must add the entire equivalence class of \( o \), denoted by \( o/ \equiv \), and hence \( \pi' = \pi \cup o/ \equiv \).

Note that while events may re-occur, each occurrence is distinct. Thus, given \( \pi \) and \( o \), if \( o \) happens then \( o \) cannot belong to \( \pi \). Hence, \( \pi \cap o/ \equiv = \emptyset \).

Adopting the notation used in [Shi97], if \( X = o/ \equiv \), we shall write \( \pi \Rightarrow X \pi' \) to indicate that \((\pi, X, \pi')\) is a step and we shall refer to it as a step from \( \pi \) to \( \pi' \) via the occurrences in \( X \). The notion of a step proves particularly useful in defining a class of acceptors for behavioural presentations. This is done in [Shi97] using asynchronous transition systems (ATS) [Shi85].

Second, we want to ensure that a behavioural presentation contains enough points to separate events which are strictly ordered or non-simultaneous. This is the replication property and is defined as follows.

**Definition 6.2.1.** If \( B = (O, I, E, \lambda) \) is a behavioural presentation, then \( B \) will be said to be replete iff whenever \( \pi_1, \pi_2 \in I \) such that \( \pi_1 \subseteq \pi_2 \) and \( o_1, o_2 \in \pi_2 \setminus \pi_1 \), then

\[
o_2 \neq o_1 \Rightarrow \exists \pi_3 \in I : (\pi_1 \subseteq \pi_3 \subseteq \pi_2) \land (o_1 \in \pi_3) \land (o_2 \notin \pi_3)
\]

In further explanation, suppose that we have points \( \pi_1 \) and \( \pi_2 \) such that \( \pi_1 \) is before \( \pi_2 \), and \( o_1, o_2 \) occurred between \( \pi_1 \) and \( \pi_2 \). Now if \( o_2 \) occurred later than or concurrently with \( o_1 \) (i.e. \( o_2 \neq o_1 \)), then the replication property says that there is a point (another possible world) \( \pi_3 \), after \( \pi_1 \) and before \( \pi_2 \), at which it is legitimate to assert that \( o_1 \) has happened but \( o_2 \) has not.

The idea of the replication property perhaps can be best illustrated by the simple example given in [Shi97]. In short, a coin is tossed (occurrence \( c \)) and then it either lands with heads on top (occurrence \( h \)) or tails on top (occurrence \( t \)). A behavioural presentation in this case would give points \( \pi_0 = \emptyset \), \( \pi_1 = \{c, h\} \), \( \pi_2 = \{c, t\} \) and that \( c < h, c < t \) and \( h \parallel t \). However, there is a possible world missing. That is, \( \pi_3 \) in which the coin has been tossed but not landed yet. That would be \( \pi_3 = \{c\} \). Therefore, \( \pi_3 \) is in between \( \pi_0 \) and \( \pi_1 \) (similarly, for \( \pi_2 \) and \( \pi_3 \)) and \( c \) has happened but \( h \) has not (similarly, \( c \)
has happened and \( t \) has not). It is situations like this that the repletion property is intended to capture. Referring back to discrete component languages (Definition 4.2.2), repletion identifies 'gaps' in the time continuum.

Essentially, a non-replete behavioural presentation will have certain points missing. The following definition gives an ordering on subsets of a behavioural presentation and, in effect, says that missing points of non-replete behavioural presentations must lie under existing points. Intuitively, in passing from a behaviour described in \( \pi_1 \) to a longer behaviour in \( \pi_2 \), so that \( \pi_1 \subseteq \pi_2 \), no elements of \( \pi_1 \) acquire new predecessors. In the sense of the following definition, \( \pi_1 \leq \pi_2 \).

**Definition 6.2.2.** Suppose that \( B = (O, \Pi, E, \lambda) \) is a behavioural presentation and that \( X, Y \subseteq O \), then we define \( X \leq Y \) if and only if

- \( X \subseteq Y \)
- \( \forall o \in Y, \forall o' \in X : o \rightarrow o' \Rightarrow o \in X \)

**Remark 6.2.1.** The relation \( \leq \) is a partial order on \( \wp(O) \).

**Proof.** Reflexivity and antisymmetry of \( \leq \) follow from reflexivity and antisymmetry of \( \subseteq \). Now suppose that \( X, Y, Z \subseteq O \) with \( X \leq Y \) and \( Y \leq Z \). Then, we may deduce that \( X \subseteq Z \). Let \( o \in X \) and \( o' \in Z \) such that \( o' \rightarrow o \). It suffices to show that \( o' \in X \). We have that \( o \in Y \) as \( X \subseteq Y \). Since \( Y \leq Z \), we may deduce that \( o' \in Y \). So, we have \( o \in X \) and \( o' \in Y \) with \( o' \rightarrow o \). Since \( X \leq Y \), we can conclude that \( o' \in X \). Hence, we have shown that \( \leq \) is also transitive, which completes the proof.

The fact that \( \leq \) is a partial order on \( \wp(O) \) can be exploited to formalise the notion that the set of behaviours has no 'gaps' in it. The potential gaps are those points given by subsets \( X \subseteq O \) such that \( X \leq \pi \), some \( \pi \in \Pi \). We accordingly define left-closed behavioural presentations as those containing all such points.

**Definition 6.2.3.** A behavioural presentation \( B = (O, \Pi, E, \lambda) \) is left-closed if and only if

\[
\forall \pi \in \Pi, \forall X \in \wp(O) : X \leq \pi \Rightarrow X \in \Pi
\]
In informally, a left-closed behavioural presentation is a behavioural presentation in which any set of occurrences that could be describing an 'earlier' portion of behaviour is itself a behaviour.

We saw that the repletion property identifies missing points. Left-closure ensures that these are included in \( \Pi \) so that a left-closed behavioural presentation is replete. It includes all 'reasonable' points. Consider the situation described earlier, where \( \pi_1 \) and \( \pi_2 \) are such that \( \pi_1 \leq \pi_2 \) and there are occurrences \( o_1, o_2 \in \pi_2 \setminus \pi_1 \) such that \( o_2 \not\succ o_1 \).

Repletion says that there must be a point \( \pi_3 \), with \( \pi_1 \subseteq \pi_3 \subseteq \pi_2 \), in which \( o_1 \) has occurred, so \( o_1 \in \pi_3 \), but not \( o_2 \), so \( o_2 \not\in \pi_3 \). Now, left closure says that a candidate for \( \pi_3 \) is the smallest point containing both \( \pi_1 \) and \( o_1 \). Since \( o_2 \not\succ o_1 \), we may deduce that this point needs to have \( \pi_1 \cup \{ o_1 \} \) as a subset and contain all its predecessors. In fact, such a set can be defined as follows.

**Definition 6.2.4.** Suppose that \( B = (O, \Pi, E, \lambda) \) is left-closed, then we define, for \( o \in O \),

\[
\downarrow o = \{ o' \in O : o' \to o \}
\]

The sets \( \downarrow o \) turn out to be the complete primes in the poset \( (\Pi, \leq) \) as shown in [Shi97].

This builds on an early result in [Shi92] (see Theorem 2.1 in [Shi92]) which allows to build a left-closed behavioural presentation from a prime algebraic and consistently complete poset together with a function that maps its prime elements onto the set of events in the corresponding behavioural presentation.

This result is significant in the construction described in the following section (Section 6.3) for associating well-behaved components with discrete behavioural presentations, and in particular in showing that a component language gives rise to a left-closed behavioural presentation (cf Proposition 6.3.4, and consequently also Proposition 6.3.1).

It might be worth pointing out that the basic properties of discrete behavioural presentations, namely consistent completeness and prime algebraicity, depend almost solely on left-closure. Evidence on this can be found in the following section.

Bringing together all above concepts we may now consider a subclass of behavioural presentations which is well-suited for describing the behaviour of discrete systems. This inspires the notion of discrete behavioural presentations as we define next.
Definition 6.2.5. A behavioural presentation $B = (O, \Pi, E, \lambda)$ will be said to be discrete if and only if, for every $\pi \in \Pi$ we have,

1. The set of equivalence classes of the elements of $\pi$ is finite

2. If $X \leq \pi$, then $X \in \Pi$

Point (1) of the above definition asserts that only a finite number of occurrences may take place within finite time. In mathematical terms, $\forall \pi \in \Pi : |\pi/ \equiv | < \infty$. It relates to finiteness and with regard to Definition 4.2.2 of discrete component languages, it excludes infinite ascending and descending chains of occurrences of events. By examining Definition 6.2.1 it can be seen that a non-replete behavioural presentation may have some points missing. Definition 6.2.3 says that such points must lie under existing points. Point (2) of the above definition then, guarantees inclusion of those points and thus ensures that there will be no points missing. With regard to Definition 4.2.2 of discrete components, this ensures that there are no 'gaps' in the time continuum.

Note that point (2) essentially reflects Definition 6.2.3 so that a discrete behavioural presentation is one that is left-closed and additionally satisfies the finitary condition (point (1) of Definition 6.2.5). Finally, note that since $\emptyset \leq \pi$, for all $\pi \in \Pi$ it follows, again from point (2) of Definition 6.2.5, that discrete behavioural presentations have bottom elements. This is the initial point in which nothing has happened yet, in the sense of discrete component languages (Definition 4.2.2).

To clarify the terminology used, and avoid confusion, discreteness and local left-closure in a component language are defined as two separate properties whereas discreteness in behavioural presentations includes the notion of left-closed behavioural presentations. Hence, a component language can be discrete but not locally left-closed (in which case it is not normal and the corresponding component is not well-behaved); a discrete behavioural presentation is always left-closed.

The obvious connotations of the naming are intentional. We will see that the normality property of component languages manifests itself in discrete behavioural presentations.
6.3 From Component Languages to Behavioural Presentations

In this section, we relate (the language part of) a component, described in Chapter 3, to a behavioural presentation, thereby building a bridge between algebraic and order-theoretic representation of component behaviour. In particular, we describe a construction that translates the language of a well-behaved component into a discrete behavioural presentation.

The translation builds on the result discussed earlier (with regard to Definition 6.2.4) which says that a prime algebraic and consistently complete poset gives rise to a left-closed behavioural presentation whose set of occurrences is the set of prime elements of the poset. This key idea appeared in [Shi92] and underlines the relationship between language-theoretic objects (such as vector languages) and order-theoretic objects (such as behavioural presentations).

A component language together with coordinate-wise prefix ordering is a poset and hence we may exploit this relationship in associating components with behavioural presentations. Our intention is to obtain an event-oriented description of component behaviour in terms of the temporal relations between events on component interfaces.

The presentation of the overall construction has been restricted to the key technical results that enable this association. In particular, we describe how a component language can be mapped onto a quadruple that mirrors a behavioural presentation and then characterise prime elements of a component language. This is illustrated with the example of the thesis in Section 6.4. The full development together with the complete proofs can be found in [SM04b].

We have seen (Section 4.2) that the language of a well-behaved component is discrete and locally left-closed. In mapping the language onto a discrete behavioural presentation the main challenge lies with left-closure. The additional finiteness constraint (point (1) of Definition 6.2.5) that makes a left-closed behavioural presentation discrete, can be guaranteed by discreteness of the component language (Definition 4.2.2).

In the study of left-closed behavioural presentations in [Shi97], which provides useful
insights on the subject matter, it is shown that if \( B = (O, \Pi, E, \lambda) \) is a left-closed behavioural presentation, then the partially ordered set \((\Pi, \leq)\) is prime algebraic and consistently complete, with the elements of \( \downarrow o \) as primes, where \( \downarrow o = \{ o' \in O : o' \rightarrow o \} \) as before.

Let us briefly recall some basic definitions from order theory (see e.g. [DP90]).

An element \( x \) of a partially ordered set is \textit{prime} if, whenever \( U \subseteq X \) and \( x \leq \bigcup U \in X \), then \( x \leq u \), for some \( u \in U \). The set of all primes of \((X, \leq)\) will be denoted by \( \text{Pr}(X) \).

A partially ordered set \((X, \leq)\) is \textit{prime algebraic} if whenever \( x \in X \), then \( x = \bigcup U \), where \( U = \{ u \in X : u \in \text{Pr}(X) \land u \leq x \} \) and \( \bigcup U \) denotes the least upper bound of the set \( U \). The least upper bound of \( U \subseteq X \), denoted by \( \bigcup U \), if it exists, is the least element \( x \in X \) such that \( u \leq x \), all \( u \in U \).

Finally, a partially ordered set \((X, \leq)\) is \textit{consistently complete} if whenever \( U \subseteq X \) and \( x \in X \) with \( u \leq x \), all \( u \in U \), then \( \bigcup U \in X \).

Given that we want to start with a component language (which is in effect a partially ordered set \((V, \leq)\)) and end up with a left-closed behavioural presentation, we are actually interested in working in the opposite direction: what is required of a partially ordered set if it is to give rise to a left-closed behavioural presentation.

**Proposition 6.3.1.** Suppose that \((X, \leq)\) is prime algebraic and consistently complete with primes \( \text{Pr}(X) \) and that \( \lambda : \text{Pr}(X) \rightarrow E \), then if we define

- \( O_X = \text{Pr}(X) \)
- \( \Pi_X = \{ \pi_x \in X \} \) where \( \pi_x = \{ u \in \text{Pr}(X) : u \leq x \} \)
- \( E_X = E \)
- \( \lambda_X = \lambda \)

then \( B_X = (O_X, \Pi_X, E_X, \lambda_X) \) is a left-closed behavioural presentation.

**Proof.**
We have that $\bigcup_{n \in \Pi_X} \pi_n = O_x$ by definition of $\Pi_X, O_X$, so $B_X$ is a behavioural presentation. We now need to show it is left-closed.

Suppose that $x \in X$ and $U \subseteq X$ such that $U \leq \pi_x$. We shall show that $U \in \Pi_X$. By definition of $\pi_x$, we have that $u \leq x$, each $u \in U$. Now, by consistent completeness of $(X, \leq)$, we may deduce that $y = \bigcup U$ is an element of $X$. It suffices to show that $\pi_y = U$.

Suppose that $u \in \pi_y$. By definition of $\pi_y$, we have that $u$ is prime and $u \leq y = \bigcup U$. By definition of prime elements in $(X, \leq)$, we may deduce that $u \leq v$, some $v \in U$. But now, $u \in \pi_y \subseteq \pi_x, v \in U, u \leq v$ and $U \leq \pi_x$, so $u \in U$.

Conversely, suppose that $u \in U$. Then, $u \leq \bigcup U = y$ and $u$ is a prime, so $u \in \pi_y$.

Hence, we have shown that $\pi_y = U$, so $U \in \Pi_X$, which completes the proof. □

It can be seen that in order to associate components with behavioural presentations we need to characterise primes in component languages and prove prime algebraicity and consistent completeness.

Before embarking on this, let us first describe a construction that maps a component language onto a quadruple $(O_Y, \Pi_Y, E_Y, \lambda_Y)$ which mirrors the behavioural presentation of Proposition 6.3.1.

The construction is along the lines of that proposed in [MSKF03, MKS05]. The only difference is with the way we define the occurrence function. In [MSKF03] it carries information about occurrences, events, operations and interfaces explicitly while in this text it conveys the information (about interfaces) implicitly. The reason for choosing the latter option in this text, is that the mapping onto a component language is then slightly more straightforward.

We start by exploiting the basic properties of the associated order theoretic structures. Recall that the relation 'covers' (see Definition 4.2.5) provides an ordering among vectors of a component language, in which one covers the other, allowing no other vector to exist in between them. For $y \in V$, we define

$$ \text{cov}_V(y) = \{ u \in V : u \prec y \} $$
This set contains all component vectors that are related to \( y \) by '<'. The following definition will lead to a characterisation of primes in a component language.

**Definition 6.3.1.** If \( c = (\Sigma, V) \) is a well-behaved component, then we shall say that \( z \in V \) is primal if there exists exactly one \( y \in V \) such that \( y < z \).

We shall write \( \text{prml}(V) \) for the set of all primal vectors in \( V \). Put formally,

\[
\text{prml}(V) = \{ y \in V : |\text{cov}(y)| = 1 \}
\]

where, if \( X \) is a set, then \( |X| \) denotes the cardinality of \( X \).

If \( y \in \text{prml}(V) \), then we define \( \text{base}(y) \) to be the unique element of \( \text{cov}(y) \). Put formally,

\[
y = \text{base}(y) \iff y \in \text{prml}(V) \land y \in \text{cov}(y)
\]

We pause to make the observation that, given our interest in well-behaved components, \( V \) is discrete and thus, the elements of \( \text{prml}(V) \) are precisely the complete primes of the partially ordered set \( (V, \leq) \) in the usual sense of domain theory. We return to this issue once we are done with the construction.

In terms of the behaviour of the corresponding component, each primal vector \( z \in \text{prml}(V) \) represents behaviour in which a call to operation (to be more precise, this can also be a set of simultaneous events) has occurred, during the course of behaviour since that described by \( \text{base}(y) \). These are captured by the corresponding column vector \( e \), which takes \( \text{base}(y) \) and 'stretches it up' to \( y \). Recall that \( e \in E_\Sigma \) (see Definition 4.2.6) represents a simultaneity class of events, those events that appear on distinct (non-empty) coordinates of \( e \). We accordingly associate primes in \( V \) with simultaneity classes of event occurrences, as we define next.

**Definition 6.3.2.** Suppose that \( c = (\Sigma, V) \) is a well-behaved component and let

\[
O_V = \{ y \in V : y \in \text{prml}(V) \}
\]

We define a function \( \lambda_V : O_V \to E_\Sigma \) by \( \lambda_V(y) = e \) if \( y \ll y \) and \( y/x = e \).

The set \( O_V \) comprises the possible occurrences of events in the behaviour of a component. As for the occurrence function, it associates occurrences with the events of which
they are occurrences; if \( \lambda_Y(o) = e \), then \( o \) is the occurrence of an event \( e \), which takes place during behaviour described by \( Y \) and since that already described by \( \text{base}(y) \). The function \( \lambda_Y \) conveys information about the last occurrence of an operation call at interface \( i \), where \( i \in I_Y \) is such that \( e(i) \neq \Lambda \). In effect, we isolate the last call out of the sequence of calls to operations on interface \( i \) of component vector \( y \). We also need to define a set of points.

**Definition 6.3.3.** For \( y \in V \), we define

\[
\pi_y = \{ u \in O_Y : u \leq y \}
\]

The set \( \pi_y \) comprises the set of all occurrences of events during the component behaviour described by \( y \). The set of all sets \( \pi_y \) for \( y \in V \), constitutes the set of points \( \Pi_Y \), hence

\[
\Pi_Y = \{ \pi_y : y \in V \}
\]

Now we may proceed to characterise primes in normal component languages and prove prime algebraicity and consistent completeness. Then, using Proposition 6.3.1 we may associate well-behaved components with left-closed behavioural presentations.

First we formally identify primes in a discrete component language as its primal vectors. The intuition is the following.

A component vector is primal (Definition 6.3.1) if it has a unique other vector immediately beneath it. Considering the order structure of a component language described in Chapter 4, the only case of a vector having more than one other vector immediately beneath it is if it sits at the top of a diamond. We have seen that such diamonds arise as a result of concurrency and effectively reflect the characteristic structure of a finite lattice (e.g. see Figure 4.10).

Now consider the set \( D = \{ a, y, z, w \} \) equipped with an ordering relation \( \leq \) such that \( z \leq a, z \leq y, y \leq w \) and \( x \leq w \). The order structure of the partially ordered set \( (D, \leq) \) exhibits the characteristic structure of a finite lattice. For the element \( w \) sitting at the top of the diamond, we have that \( w \leq z \cup y \) but \( w \not\leq z \) and \( w \not\leq y \). Thus, \( w \) cannot be a prime in \( (D, \leq) \). For the element \( z \) sitting in the middle of the diamond, we have
that $x \leq \bigcup D$ and $x \leq z$ so $x$ is a prime in $(D, \leq)$. Similarly, for $y$. Finally, $z$ is the least element in $(D, \leq)$ and like all least elements cannot be a prime (it is the least upper bound of $\emptyset$ but there is no element in $\emptyset$ that is related to $z$ by $\leq$).

Therefore, the notion of primal vectors is understood as an interpretation of primes specifically for languages of well-behaved components. This is established in the following proposition.

**Proposition 6.3.2.** Suppose that $c = (D, V)$ is a well-behaved component and $z \in V$ with $z \notin A$. Then, the following are equivalent:

1. $z$ is prime
2. $z$ is primal
3. There exists unique $u \in V$ and unique $e \in D$ such that $u < z$ and $z = u \cdot e$

**Proof.**

We need to show that (1) $\iff$ (2) and that (2) $\iff$ (3).

- (1) $\iff$ (2)
  - (1) $\implies$ (2)
    Assume that $z$ is not primal. Since $A < z$, there exists at least one $u \in V$ such that $A < u < z$. Indeed, the set $Y = \{y \in V : A < y < z\}$ is non-empty (it contains $A$) and finite, and therefore contains a maximal element $u$, which satisfies the claim of existence of such $u \in V$.

    Since $z$ is not primal, there must exist at least two such vectors $u_1, u_2 \in V$. Hence, $u_k < z$, each $k$. By Lemma 4.2.1 (2), $u_1 \cup u_2 = z$. But now, $z$ is prime, which implies that $z \leq u_k$, some $k$, which is a contradiction (because $u_k < z$ implies that $z \leq u_k$, either $k$). Therefore, $z$ is primal.

  - (2) $\implies$ (1) (Sketch)

    We use Lemma 3.2 in [SM04a], which says that whenever $V$ is discrete and
6.3. From Component Languages to Behavioural Presentations

\[ z \in V \text{ such that } u_1, u_2 \in V, \text{ with } z = u_1 \cup u_2 \in V \text{ then either } z = u_1 \text{ or } z = u_2, \text{ then we may deduce that } z \text{ is prime in } V. \]

Suppose that \( z = u_1 \cup u_2, \) with \( u_1, u_2, u_1 \cup u_2 \in V. \) For a contradiction with the lemma stated above, assume that \( z \neq u_k, \) each \( k. \) As in the proof of (1) \( \implies (2), \) there exists \( u_1, u_2 \in V \) such that \( u_k \leq u_k \cup z, \) each \( k. \) Since \( z \) is prime, \( u_1 = u_2 = u, \) say. But now, \( u_1 \cup u_2 \leq u \cup z, \) giving the contradiction \( u_1 \cup u_2 < z \) (because we started with \( u_1 \cup u_2 = z). \)

Hence, we have shown that \( z = u_k, \) some \( k, \) and now Lemma 3.2 in [SM04a] gives that \( z \) is prime.

- \( (2) \iff (3) \)

- \( (2) \implies (3) \)

This is Lemma 4.2.2, in Section 4.2, Chapter 4.

- \( (3) \implies (2) \)

Suppose that (3) holds. Let \( u_1, u_2 \in V, \) such that \( u_1, u_2 \leq z. \) By similar reasoning to that used in \( (2) \implies (1), \) we may argue that \( z = u_k \cup u_k, \) each \( k. \)

Now, \( u_1 = u_2 \) follows from the fact that (3) holds, giving that \( z \) is primal.

\[ \square \]

**Example 6.3.1.** Consider the component language of the CMenu component of the example of Section 4.4, and in particular its order structure as given in Figure 4.10.

The primal vectors in this case are

\[
(A, b1, A), (a1, A, A), (A, b1b2, A), (A, b1b3, A), (a1a2, A, A), (a1a2, A, c1)
\]

It can be seen that these vectors satisfy (3) of Proposition 6.3.2.

We now come to the main result that will allow us to use a behavioural presentation to model the observable behaviour of a component. This is an adaptation of the result in [Shi92] for component languages. The full proof is given in a technical report by Shields and Moschyiannis [SM04b].
Proposition 6.3.3. Suppose that $c = (\Sigma, V)$ is a well-behaved component. Then, $V$ is prime algebraic with the primal elements as primes.

Proof. (Sketch)

Let $u \in V$, and define $Pr(u) = \{z \in Pr(V) : z \preceq u\}$. We show that $\bigcup Pr(u) = u$ by proving $\bigcup Pr(u) \preceq u$ and then the reverse inequality. See Proposition 4.1 in [SM04a] for the complete proof. □

Proposition 6.3.4. Suppose that $c = (\Sigma, V)$ is a well-behaved component. Then, in the notation of Proposition 6.3.1, $B_V = (O_V, \Pi_V, E_V, \lambda_V)$ where

- $O_V = \text{prml}(V)$
- $\Pi_V = \{\pi_u \in O_V\}$, where $\pi_u = \{u \in O_V : u \preceq v\}$
- $E_V = E_\Sigma$
- $\lambda_V : O_V \to E_\Sigma$ given by $\lambda_V(v) = e$ if $u \preceq u$ and $v/u = e$

is a left-closed behavioural presentation.

Proof.

We need to show that $B_V$ is prime algebraic and consistently complete with the primal elements as primes.

By Proposition 6.3.3, we have that $(V, \preceq)$ is prime algebraic with the primal elements as primes. By discreteness, together with the fact that $X \subseteq V$ and $v \preceq x$, all $x \in X$ implies that $X$ is finite and $\bigcup X \in V$, we have that $(V, \preceq)$ is consistently complete. Thus, by Proposition 6.3.1, $B_V$ is a left-closed behavioural presentation. □

To sum up, we wanted to obtain a discrete behavioural presentation for a well-behaved component. We have seen that a discrete behavioural presentation (Definition 6.2.5) is one which is left-closed (condition (2) of the definition) and the set of occurrences in each of its points is finite (condition (1)). Discreteness of the component language
6.3. From Component Languages to Behavioural Presentations

$V$ of a well-behaved component guarantees the latter. To establish condition (2) it turned out (Proposition 6.3.1) that it suffices to characterise primes in $(V, \leq)$ and prove that it is prime algebraic and consistently complete. Primes in a component language were characterised in Proposition 6.3.2 using the notion of primal vectors (Definition 6.3.1). Based on this characterisation, Proposition 6.3.3 establishes that $(V, \leq)$ is prime algebraic. Consistent completeness of $(V, \leq)$ draws upon discreteness of $V$ as shown in Proposition 6.3.4 which establishes that the language $V$ of a well-behaved component gives rise to a left-closed behavioural presentation. This addresses condition (2) of the definition of discrete behavioural presentations. Thus, a well-behaved component can be associated with a discrete behavioural presentation.

Our preoccupation with locally left-closed component languages and left-closed behavioural presentations, which form a large chunk of this connection, can be explained more clearly as follows.

The local left-closure property of components (Definition 4.2.3) takes up on ideas of left-closed behavioural presentations and in particular, of the ordering amongst subsets of the set of occurrences in a behavioural presentation, given in Definition 6.2.2. Consider the component $c = (\Sigma, V)$, where $I_\Sigma = \{1, 2\}$, whose component language is given by

$$V = \{(\Lambda, \Lambda), (aa, \Lambda), (\Lambda, bb), (aa, bb)\}$$

It can be shown that $V$ constitutes a finite lattice, so $V$ is discrete. However, the corresponding behavioural presentation would have the counterintuitive property that although four operation calls have occurred on component interfaces there are only two elements in $O_V$ to describe them, namely $(aa, \Lambda)$ and $(\Lambda, bb)$. This is because the two primal vectors in $V$ represent the occurrence of the second of the operation calls on each interface. It is for this reason that we require local left-closure of a component language - local left-closure of $V$ in this case would entail the inclusion of vectors $(a, \Lambda)$ and $(\Lambda, b)$ in $V$. With regard to the notion of left-closed behavioural presentations, occurrence of $a$ in $(a, \Lambda)$ and $b$ in $(\Lambda, b)$ describe some 'earlier' fragment of behaviour and thus, should be considered separately as behaviours on their own. Including $(a, \Lambda)$ for instance, does just that; it describes behaviour in which we have the first occurrence of an $a$ on interface $1$. Similarly, for $(\Lambda, b)$. 
6.4 Illustration by example

In this section, we apply the formal construction introduced in this chapter to obtain an event-oriented description of the CMenu component of the case study considered in Sections 4.4, 5.4.

Recall that the formal description of the CMenu component is given by $c_M = (\Sigma_M, V_M)$ where $\Sigma_M = (P_M, R_M, \beta_M)$ is the component signature and $V_M$ is the component language which comprises the following component vectors:

$$V_M = \{ (\Lambda, \Lambda, \Lambda), (a_1, \Lambda, \Lambda), (a_1, b_1, \Lambda), (a_1 a_2, \Lambda, \Lambda), (a_1, b_1 b_2, \Lambda), (a_1, b_1 b_3, \Lambda),$$

$$\ldots, (a_1 a_2, \Lambda, c_1), (a_1 a_2, b_1, \Lambda), (a_1 a_2, b_1, c_1), (a_1 a_2, b_1 b_2, c_1), (a_1 a_2, b_1 b_3, c_1) \} \}

The first step is to determine the set $\text{cov}(y)$, for each component vector $y \in V_M$.

$$\text{cov}(\Lambda, \Lambda, \Lambda) = \emptyset$$

$$\text{cov}(a_1, \Lambda, \Lambda) = \{(\Lambda, \Lambda, \Lambda)\}$$

$$\text{cov}(a_1, b_1, \Lambda) = \{(a_1, \Lambda, \Lambda)\}$$

$$\text{cov}(a_1 a_2, \Lambda, \Lambda) = \{(a_1, \Lambda, \Lambda)\}$$

$$\text{cov}(a_1, b_1 b_2, \Lambda) = \{(a_1, b_1, \Lambda)\}$$

$$\text{cov}(a_1, b_1 b_3, \Lambda) = \{(a_1, b_1, \Lambda)\}$$

$$\text{cov}(a_1 a_2, \Lambda, c_1) = \{(a_1 a_2, \Lambda, \Lambda)\}$$

$$\text{cov}(a_1 a_2, b_1, \Lambda) = \{(a_1, b_1, \Lambda), (a_1 a_2, \Lambda, \Lambda)\}$$

$$\text{cov}(a_1 a_2, b_1, c_1) = \{(a_1 a_2, b_1, \Lambda), (a_1 a_2, \Lambda, c_1)\}$$

$$\text{cov}(a_1 a_2, b_1 b_2, c_1) = \{(a_1 a_2, b_1, c_1)\}$$

$$\text{cov}(a_1 a_2, b_1 b_3, c_1) = \{(a_1 a_2, b_1, c_1)\}$$

We may now identify the primal vectors in $V_M$. These are vectors whose corresponding set $\text{cov}(y)$ contains a unique vector. Hence,

$$\text{prml}(V) = \{(a_1, \Lambda, \Lambda), (a_1, b_1, \Lambda), (a_1 a_2, \Lambda, \Lambda), (a_1, b_1 b_2, \Lambda), (a_1, b_1 b_3, \Lambda),$$

$$\ldots, (a_1 a_2, \Lambda, c_1), (a_1 a_2, b_1 b_2, c_1), (a_1 a_2, b_1 b_3, c_1)\}$$
These vectors describe the behaviour of the CMenu component that arises from each occurrence of an event on its interfaces.

Next, the primal vectors are associated with the occurrences of events which refer to the last call to an operation during the fragment of behaviour described by each primal vector.

\[
\begin{align*}
o_1 &= (a_1, A, A) \\
o_2 &= (a_1, b_1, A) \\
o_3 &= (a_1 a_2, A, A) \\
o_4 &= (a_1, b_1 b_2, A) \\
o_5 &= (a_1, b_1 b_3, A) \\
o_6 &= (a_1 a_2, A, c_1) \\
o_7 &= (a_1 a_2, b_1 b_2, c_1) \\
o_8 &= (a_1 a_2, b_1 b_3, c_1)
\end{align*}
\]

Hence, the set of occurrences \( O_{VM} \) is given by

\[
O_{VM} = \{ o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8 \}
\]

and comprises all possible occurrences of events in the behaviour of the CMenu component resulting from its participation in the tuning task, as described in Sections 4.4, 5.4.

Next, the occurrence function \( \lambda_V \) is used to associate each occurrence with the event it corresponds to. Hence, \( \lambda_{VM} \) determines which operation call, and on which interface, each occurrence refers to.

\[
\begin{align*}
\lambda_{VM}(o_1) &= (a_1, A, A) \\
\lambda_{VM}(o_2) &= (A, b_1, A) \\
\lambda_{VM}(o_3) &= (a_2, A, A) \\
\lambda_{VM}(o_4) &= (A, b_2, A)
\end{align*}
\]
\lambda_{V_M}(o_5) = (\Lambda, b3, \Lambda) \\
\lambda_{V_M}(o_6) = (\Lambda, \Lambda, c1) \\
\lambda_{V_M}(o_7) = (\Lambda, b2, \Lambda) \\
\lambda_{V_M}(o_8) = (\Lambda, b3, \Lambda)

Notice that \( \lambda_{V_M} \) returns the column vector representing the event associated with each occurrence, or in other words, with each primal vector in \( V_M \). The occurrence function is to be read in conjunction with the corresponding occurrence or primal vector given earlier. For instance, \( \lambda_{V_M}(o_7) = (\Lambda, b2, \Lambda) \), which can also be written as \( \lambda_{V_M}((a1a2, b1b2, c1)) = (\Lambda, b2, \Lambda) \), says that the primal vector \((a1a2, b1b2, c1)\) describes the occurrence of a call to operation \( b2 \) on interface IFineTune. Effectively, this information is conveyed by its association with \( g = (\Lambda, b2, \Lambda) \in E_{BM} \) which identifies \( b2 \) as the last operation call (event) in the fragment of behaviour described by \((a1a2, b1b2, c1)\).

It might be worth pointing out that the example demonstrates that the same event may be associated with different occurrences. Consider \( \lambda_{V_M}(o_7) \) again, which implies that \( o_7 \) is the occurrence of call to operation \( b2 \) on IFineTune. This is also the case for \( \lambda_{V_M}(o_4) \) which is also associated with \( b2 \) since \( \lambda_{V_M}(o_4) = (\Lambda, b2, \Lambda) \). However, in \( \lambda_{V_M}(o_4) \) the call to operation \( b2 \) is associated with \( o_4 = (a1, b1b2, \Lambda) \) which means that \( o_4 \) refers to occurrence of \( b2 \) on IFineTune when \( a1 \) and \( b1 \) have taken place earlier whereas in \( \lambda_{V_M}(o_7) \) the event \( b2 \) is associated with \( o_7 = (a1a2, b1b2, c1) \) and thus refers to occurrence of \( b2 \) on IFineTune but only after the component has experienced calls to operations \( a1, a2, b1 \) and \( c1 \). In terms of what we saw in describing behavioural presentations (Section 6.1), these two occurrences of \((\Lambda, b2, \Lambda)\) take place in different "possible worlds".

Now based on Definition 6.3.3, we may obtain the set of occurrences of calls to operations that have taken place during the fragment of behaviour described by each vector in \( V_M \).

\( \pi(\Lambda, \Lambda, \Lambda) = \emptyset \)

\( \pi(a1, \Lambda, \Lambda) = \{o_1\} \)

\( \pi(a1, b1, \Lambda) = \{o_1, o_2\} \)
6.4. Illustration by example

\[ \pi_{(a1a2, A, A)} = \{o_1, o_3\} \]
\[ \pi_{(a1, b1b2, A)} = \{o_1, o_2, o_4\} \]
\[ \pi_{(a1, b1b3, A)} = \{o_1, o_2, o_5\} \]
\[ \pi_{(a1a2, A, c1)} = \{o_1, o_3, o_6\} \]
\[ \pi_{(a1a3, A, A)} = \{o_1, o_2, o_3\} \]
\[ \pi_{(a1a2, b1, A)} = \{o_1, o_2, o_3, o_6\} \]
\[ \pi_{(a1a2, b1b2, c1)} = \{o_1, o_2, o_3, o_6, o_7\} \]
\[ \pi_{(a1a2, b1b3, c1)} = \{o_1, o_2, o_3, o_6, o_8\} \]

For instance, \( \pi_{(a1a2, A, c1)} = \{o_1, o_3, o_6\} \) contains occurrence of call to operation \( a1 \) on interface \( \text{ISe}arch\text{Fr} \) (i.e. \( o_1 \)), occurrence of call to operation \( a2 \) on interface \( \text{ISe}arch\text{Fr} \) (i.e. \( o_3 \)) and occurrence of call to operation \( c1 \) on interface \( \text{IDet}ect\text{S}ignal \) (i.e. \( o_6 \)). These three event occurrences comprise the fragment of behaviour of the component described by the component vector \( (a1a2, A, c1) \in V_M \). The union of the above sets comprise the set of points \( \Pi_{V_M} \).

Referring back to Definition 6.1.1, we have constructed the sets \( O_{V_M} \) and \( \Pi_{V_M} \) for the \( \text{CMenu} \) component. The quadruple \( O_{V_M}, \Pi_{V_M}, E_{\Sigma_M}, \lambda_{V_M} \) where

\[ E_{\Sigma_M} = (a1, A, A), (A, b1, A), (a2, A, A), (A, b2, A), (A, b3, A), (A, A, c1) \]

and \( \lambda_{V_M} : O_{V_M} \rightarrow E_{\Sigma_M} \) is as given earlier, is a behavioural presentation for \( V_M \) and in particular, a discrete behavioural presentation since \( \text{CMenu} \) is well-behaved.

The relationships between any pair of occurrences in \( O_{V_M} \) are given in Table 6.1 using the temporal relations of Definition 6.1.2.

Consider for example the relation between \( o_3 \) and \( o_6 \). Occurrence \( o_6 \) refers to operation \( c1 \) on \( \text{IDet}ect\text{S}ignal \), but only when calls to operations \( a1 \) and then \( a2 \) on \( \text{ISe}arch\text{Fr} \) have preceded it. Occurrence of a call to operation \( a2 \) on \( \text{ISe}arch\text{Fr} \) after \( a1 \) has occurred on the same interface is precisely occurrence \( o_3 \). Thus, \( o_3 \) strictly precedes \( o_6 \) since \( a1 \) and \( a2 \) must have occurred at interface \( \text{ISe}arch\text{Fr} \) (this is \( o_3 \)) before \( c1 \) can occur at interface \( \text{IDet}ect\text{S}ignal \) (this is \( o_6 \)).
Table 6.1: Ordering on occurrences of events of the CMenu component

<table>
<thead>
<tr>
<th>for o₁:</th>
<th>for o₂:</th>
<th>for o₃:</th>
<th>for o₄:</th>
<th>for o₅:</th>
<th>for o₆:</th>
<th>for o₇:</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁ &lt; o₂</td>
<td>o₂ ‖ o₃</td>
<td>o₃ ‖ o₄</td>
<td>o₄ ‖ o₅</td>
<td>o₅ ‖ o₆</td>
<td>o₆ &lt; o₇</td>
<td>o₇ ‖ o₈</td>
</tr>
<tr>
<td>o₁ &lt; o₃</td>
<td>o₂ &lt; o₄</td>
<td>o₃ &lt; o₅</td>
<td>o₄ &lt; o₆</td>
<td>o₅ &lt; o₇</td>
<td>o₆ &lt; o₈</td>
<td></td>
</tr>
<tr>
<td>o₁ &lt; o₄</td>
<td>o₂ &lt; o₅</td>
<td>o₃ &lt; o₆</td>
<td>o₄ &lt; o₇</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o₁ &lt; o₅</td>
<td>o₂ &lt; o₆</td>
<td>o₃ &lt; o₇</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o₁ &lt; o₆</td>
<td>o₂ &lt; o₇</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o₁ &lt; o₇</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>o₁ &lt; o₈</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A behavioural model of the CMenu component can be seen in Figure 6.1 which depicts the temporal relations among occurrences of events, as these are experienced by the component on its interfaces. Relations between events are given along both dimensions, horizontal and vertical, of the graph. Arrows between occurrences indicate time ordering. Time is understood, as usual, to progress top-down the page and thus arrows may appear only on the vertical dimension.

Figure 6.1: Behavioural presentation model for the CMenu component

The behaviour of the CMenu component at its interfaces, its observable behaviour, is dominated by the ‖ relation. This is mostly due to the fact that the CMenu component
establishes communication with the user, through its provided interfaces IFineTune and ISearchFre. The user makes a number of (conscious) choices in interacting with the Manual Store option of the TV men. These choices are interpreted by the CMenu component specification as determining a particular behaviour / response (essentially, a path along the diagram of Figure 4.10) out of all the possible behaviours the CMenu component is intended to exhibit. Therefore, all possible choices are made available to the user nondeterministically and only when the user makes specific choices is the behaviour of the component confined to a particular response.

6.5 Concluding note

In this chapter, we considered a behavioural presentation to model the behaviour of a component at its interfaces. By defining a construction that maps all legal component behaviours onto a behavioural presentation - specifically, a discrete behavioural presentation - our formal description of a component is given in terms of the relationships between event occurrences on component interfaces.

At the centre of this formal construction is the notion of prime elements in a partially ordered set. We characterised primes in a component language in terms of the notion of primal vectors which are component vectors having a unique other vector immediately beneath them. Such vectors represent the occurrences of events on the interfaces of a component.

Behavioural presentations describe behaviour in terms of patterns of occurrences of events over time. This allows for a precise description of the relationships between events occurring on component interfaces. The resulting event-oriented description of behaviour of a component is particularly useful in determining whether the ordering of its event occurrences is respected when it is placed in a new context.

We have seen (Chapter 4) that the ordering between event occurrences on different interfaces of a component is determined by what vectors are in its component language. This is also manifested in the order theoretic structure of the associated behavioural presentation.
Consider the component language,
\[ V_1 = \{(A, A), (a, A), (A, b), (a, b)\} \]
Its primal vectors are \((a, A), (A, b)\) for which \(\lambda_{V_1}(o_1) = (a, A)\) and \(\lambda_{V_1}(o_2) = (A, b)\). The set \((\Pi_1, \leq)\) of the resulting behavioural presentation comprises four points; \(\pi_0 = \emptyset\) when nothing has happened, \(\pi_1 = \{o_1\}\) when \(o_1\) has occurred but not \(o_2\), \(\pi_2 = \{o_2\}\) when \(o_2\) has occurred but not \(o_1\), and \(\pi_3 = \{o_1, o_2\}\) when both have. We have that \(\pi_1 \leq \pi_3\) and \(\pi_2 \leq \pi_3\) and \(o_1, o_2\) are concurrent \((o_1 \land o_2)\).

Next, consider the component language
\[ V_2 = \{(A, A), (a, b)\} \]
which has one primal vector \((a, b)\) with \(\lambda_{V_2}(o) = (a, b)\). The set \((\Pi_2, \leq)\) of the resulting behavioural presentation comprises two points; \(\pi_0 = \emptyset\) when nothing has happened, \(\pi_1 = \{o\}\) when \(o\) has occurred. This is a simultaneity class of events (as given by \(e = (a, b) \in \mathcal{E}_{V_2}\)) describing that \(a\) and \(b\) occurred simultaneously.

Finally, consider the component language
\[ V_3 = \{(A, A), (a, A), (A, b)\} \]
which has two primal vectors with \(\lambda_{V_3}(o_1) = (a, A)\) and \(\lambda_{V_3}(o_2) = (A, b)\) respectively. The set \((\Pi_3, \leq)\) comprises three points; \(\pi_0 = \emptyset\) when nothing has happened, \(\pi_1 = \{o_1\}\) when \(o_1\) has occurred but not \(o_2\), and \(\pi_2 = \{o_2\}\) when \(o_2\) has occurred but not \(o_1\). We have \(\pi_0 \leq \pi_1\) and \(\pi_0 \leq \pi_2\) and \(o_1, o_2\) are mutually exclusive \((o_1 \parallel o_2)\).
Chapter 7

State-oriented Description of Component Behaviour

In the previous chapter we have seen a description of component behaviour that focuses on the patterns of occurrences of events which a component can perform. Component behaviour can additionally be viewed from a different angle, in terms of change of state resulting from events occurring on the interfaces of a component.

In this chapter, we associate the formal description of a component in our approach with a class of automata, the so-called $\Sigma$-automata. This builds on consequences of local left-closure and discreteness in component languages. We have seen that concurrency in component languages is modelled explicitly in the form of independence between (consecutive) column vectors. In order to reflect this representation of concurrency in the state-based description of component behaviour, we have to look beyond transition systems which identify concurrency with nondeterministic interleaving of events.

A more suitable model for our purposes is that of asynchronous transition systems (ATS) [Shi85, Bed88] where transitions are thought of as occurrences of events which bear a relation of independence. The idea is that if two independent events can occur one immediately after the other, then they should be able to occur with their order interchanged. This leads to 'lozenge shapes' in an ATS, which reflect, as we will see, the diamonds in the order structure of the corresponding component language (Section
Chapter 7. State-oriented Description of Component Behaviour

Apart from concurrency, we also required discreteness (this property is also related to concurrency (Section 4.3.2)) and local left-closure of component languages in our characterisation of well-behaved components (Chapter 4). Consequently, we are interested in a class of automata that determine well-behaved components only. This requires an elaboration of ATSs which results in $\Sigma$-automata.

The automata-based formalism described in this chapter builds on previous work on establishing a connection between vector languages, behavioural presentations and ATSs, found in [Shi97]. The adaptation of this work in [SM04a] for component languages shows that, based on consequences of discreteness and local left-closure, we may derive a transition relation that may be exploited in associating components with $\Sigma$-automata which are an elaboration of ATS [Shi88]. This development is outlined in [MSK05] in which we have also included early ideas on using the resulting concurrent automata to formally underpin UML state diagrams.

In this chapter we outline the development of an automata-theoretic framework for components, based on this work, and discuss how these automata-theoretic objects can be used to give a formal semantics to compound transitions in UML and subsequently represent concurrency explicitly in UML state diagrams, using existing UML notation.

In particular, we start by describing how a $\Sigma$-automaton can be derived from a well-behaved component. Also, as a result of embedding the necessary conditions in its transition structure, a $\Sigma$-automaton from initial state determines a well-behaved component. We make connections to UML2.0 concepts that allow us to use notation from UML state diagrams for a graphical representation of a $\Sigma$-automaton. Furthermore, we outline how $\Sigma$-automata are composed and examine composition of components in relation to automata. The ideas are illustrated by means of an example.

7.1 Automata for Well-Behaved Components

In this section we associate components in our approach with $\Sigma$-automata. In particular, we describe how a $\Sigma$-automaton can be derived from the language of a well-behaved
component. Our study of automata for components is outlined in [MSK05] which also contains early attempts in establishing a relation to UML state diagrams.

We begin by defining \( \Sigma\)-machines, essentially a type of transition system, which will be refined into a \( \Sigma\)-automaton. The study of the local left-closure property provides the starting point. It is a consequence of local left-closure that whenever two component vectors \( u, v \) are such that \( u < v \) and \( u < w < v \) for no component vector \( w \), then \( v = u \cdot e \) where \( e \) is a vector each of whose coordinates is either a single event or the empty sequence. We may accordingly associate each component with a transition system having vectors such as \( e \) as labels on transitions.

A few words are in order to place this observation within the formal framework for components described so far. In a normal component language \( V \) (Definition 4.2.4), a vector \( z \) extends a vector \( u \) to a vector \( v \) if \( y = u \cdot z \) and there is no other vector in \( V \) that lies strictly between \( u \) and \( v \). The latter requirement can be expressed by saying that \( y \) covers \( u \), in the sense of Definition 4.2.5. The 'continuation' \( z \) which extends \( u \) to \( v \) is defined using the right-cancellation operator of Definition 4.1.4. We have seen (Proposition 4.2.1) that in a normal component language such continuations turn out to be column vectors from \( E \) (Definition 4.2.6). This gives a transition relation which leads to the definition of the so-called \( \Sigma\)-machines.

**Definition 7.1.1.** Let \( \Sigma \) be a signature. Then, we define a \( \Sigma\)-machine to be a pair \( M = (Q, \succ) \) where

- \( Q \) is a set of states
- \( \succ \subseteq Q \times E \times Q \) is the transition relation, and we write \( q \succ e q' \) for \( (q, e, q') \in \succ \)

which satisfies:

1. \( q \succ e q_1 \land q \succ e' q_2 \land e \leq e' \Rightarrow e = e' \land q_1 = q_2 \)
2. \( q \succ e q' \land q \succ e' q' \Rightarrow e = e' \)

We also define a rooted \( \Sigma\)-machine to be a pair \( M^* = (M, q) \) where \( M = (Q, \succ) \) is a \( \Sigma\)-machine and \( q \in Q \).
We will write $q \xrightarrow{\xi} q'$ to denote that there exists $q' \in Q$ such that $q \xrightarrow{\xi} q'$.

Condition (2) of the definition imposes that there is a unique transition between any given pair of states. Note that condition (1) includes the case that $q = q'$ in which case the condition can be rewritten as $q \xrightarrow{\xi} q_1 \land q \xrightarrow{\xi} q_2 \Rightarrow q_1 = q_2$ which is the usual deterministic condition one tends to find in transition systems. The motivation behind condition (1) is twofold; first, it guarantees unambiguity and second, it relates to the point of (3) and (4) of the subsequent definition of $\Sigma$-automata (cf Definition 7.1.5) which deal with the issue of reconstructing the language of a well-behaved component (given a $\Sigma$-automaton). This is further discussed in Section 7.2.

Rooted $\Sigma$-machines, essentially $\Sigma$-machines with initial states, determine languages of vectors. This is based on the construction given in the following definition, which introduces component vectors on transitions. Effectively, these are component vectors formed by repeatedly concatenating column vectors $\xi$ that appear on transitions along a contiguous path through the graph of the machine from the initial state. In this sense, they are reminiscent of action sequences in transition systems. We shall refer to them as execution vectors. A $\Sigma$-machine determines a set of such vectors and this set comprises the vector language generated by the machine.

**Definition 7.1.2.** Suppose that $M = (Q, \xrightarrow{\cdot})$ is a $\Sigma$-machine and $q \in Q$. Define $q \rightarrow^\Sigma q'$ if

1. $q = q'$ and $\xi = \Delta_{\xi}$
2. $\xi = \zeta \xi', \xi, \xi' \in \mathbb{E}_\Sigma$, such that $q \rightarrow^\Sigma \zeta \xrightarrow{\xi} q'$, some $q \in Q$

We also define $V(M, q) = \{\xi \in \mathbb{E}_\Sigma : \exists q' \in Q, q \rightarrow^\Sigma q'\}$.

The execution vectors of a $\Sigma$-machine can be understood as describing sequences of individual transitions. To be more precise, an execution vector describes the sequence of the transitions $\xi$ out of which it is formed. In this sense, an execution vector can be understood as a compound transition in UML state diagrams [OMG04]. This will be further discussed in the sequel.
Point (1) of Definition 7.1.2 refers to internal transitions, in UML dialect [OMG04]. Not surprisingly, these become significant when we consider composition of the corresponding automata (Section 7.3).

Point (2) of Definition 7.1.2 says that, for every compound transition, there is always a state which leads to its target state via a simple transition given by a column vector, and that state is reachable from its source state (through some other compound transition). It can be seen that this may involve decomposition of a component vector into a series of concatenations with column vectors from $E_\Sigma$, as shown in [SM04a]. This is further exploited in showing that the vector language of the corresponding $\Sigma$-automaton is locally left-closed (cf Lemma 7.2.1), as part of establishing that a $\Sigma$-automaton generates the language of a well-behaved component.

Before introducing $\Sigma$-automata we describe how a $\Sigma$-machine can be derived from a well-behaved component (i.e. from a normal component language). This is done by taking component vectors in $V$ as states and defining the transition relation in a way that reflects the observation that behaviours may be seen to be built up from the empty vector by repeatedly concatenating column vectors to it (Section 3.1.2). In fact, this takes up on the ideas presented prior to defining $\Sigma$-machines. It is put formally in the following definition.

**Definition 7.1.3.** Suppose that $c = (\Sigma, V)$ is a well-behaved component, then we define $M_c = (V, \rightarrow_V)$ where

$$u \rightarrow^V v \iff u \cdot u \cdot v = c$$

We also define $M_c^* = (M_c, A_\Sigma)$

The subscript $V$ will be dropped when the language is clear from context.

Note that $v/u \in E_\Sigma$, whenever $u \cdot u$, by Proposition 4.2.1 so the definition makes sense. This construction gives a $\Sigma$-machine as shown in the following proposition. Moreover, it can be shown that the vector language generated by $M_c$ from initial state $A_\Sigma$ determines the same component $c = (\Sigma, V)$ using the execution vectors of Definition 7.1.2. In our notation this is expressed as $V(M_c^*) = V$.

**Proposition 7.1.1.** Suppose that $c = (\Sigma, V)$ is a well-behaved component, then
1. \( M_e = (V, \succ) \) is a \( \Sigma \)-machine

2. \( V(M_e^*) = V \)

**Proof.**

For (1), we check the conditions of Definition 7.1.1.

For Definition 7.1.1(1), if \( y \succ^x u_1 \), then by Definition 7.1.3 we have \( u \succ u_1 \) and \( u_1 = u \cdot e \). Similarly, if \( u \succ^x u_2 \), we have \( u \succ u_2 \) and \( u_2 = u \cdot e' \). Since \( e \leq e' \), we have \( u_1 = u \cdot e \leq u \cdot e' = u_2 \). Hence, \( u < u_1 \leq u_2 \). As \( u \succ u_2 \), we must have \( u_1 = u_2 \). This implies that \( u \cdot e = u \cdot e' \). Hence, we must have \( e = e' \).

For Definition 7.1.1(2), if \( x \succ^y y \) and \( e \succ^{x'} y \), then by Definition 7.1.3 we have \( x \cdot e = y = x \cdot e' \) and thus, \( e = y/x = e' \).

For (2), see proof of Proposition 5.2 in [Shi05]. □

We are now set to refine \( \Sigma \)-machines to \( \Sigma \)-automata. We have seen that a well-behaved component can be turned into a \( \Sigma \)-machine. This construction provides a (primitive) state-based description of a component, but does not have the appropriate depth of expressiveness to deal with local left-closure and discreteness, and consequently also concurrency, in the related machines. These are central aspects of our formal approach to the analysis of component behaviour and therefore, we would like a class of automata that generate only discrete and locally left-closed component languages. The transition structure of a \( \Sigma \)-machine needs to be constrained accordingly.

The required constraints are presented in this section and further motivated by the development in the next section (Section 7.2) which is concerned with ensuring that \( \Sigma \)-automata only generate languages of well-behaved components. Furthermore, we have seen that concurrency is intrinsic to the discreteness property of component languages. In order to express concurrency explicitly in this context, where the same events may sometimes be concurrent and sometimes not, we will need to consider independence between component vectors (Definition 4.3.1) and determine its relationship to the transition structure of \( \Sigma \)-machines (cf Definition 7.1.4). This results in additional constraints on \( \Sigma \)-machines, leading to the definition of \( \Sigma \)-automata.
Independence between component vectors captures the fact that the behaviours described by each vector are not causally related or mutually exclusive and can take place independently. Effectively, the independence relation implies that behaviours which may occur concurrently engage distinct interfaces of the component in question. In short, \( u \text{ ind } v \) means that \( u, v \) describe behaviour of the component in which unordered events occurred on different interfaces of the component. It is important to note that independence is a prerequisite for concurrency, but independence alone does not guarantee concurrency - there is the additional requirement that the column vectors in question occur consecutively. This was also discussed in Section 4.3.2. The following definition takes into account both requirements for concurrency and formulates the property in terms of the transition structure of \( \Sigma \)-machines.

**Definition 7.1.4.** Suppose that \( M = (Q, \succ) \) is a \( \Sigma \)-machine, then we define a relation \( I^M \subseteq Q \times E_2 \times E_2 \), and we write \( e_1 I^M e_2 \) for \((q, e_1, e_2) \in I^M \), by

\[
e_1 I^M e_2 \iff e_1 \text{ ind } e_2 \land (\exists q_1, q_2, q' \in Q : q \succ q_1 \land q \succ q_2 \Rightarrow q \succ q')
\]

We shall, as usual, drop the superscript \( M \) when it is clear from context.

The relation \( I^M_q \) defines local concurrency. This should become clear in the example below which also highlights the subtle difference between \textit{ind} and \( I^M_q \).

**Example 7.1.1.** Consider the well-behaved component of Figure-7.1 which does \( b \)'s on its one port and \( a \)'s on the other. Discreteness is clear from the familiar diamond shape which shows it to be a (finite, distributive) lattice while local left-closure is clear by inspection.

In the machine derived from the well-behaved component of Figure-7.1, we have, for example, \((A, A) \succ (A, a) \succ (b, a) \succ (b, A)\) and also that \((b, a) \succ (b, a) \succ (b, b) \succ (b, b) \succ (b, a) \succ (b, a)\).

The column vectors \((b, A)\) and \((A, a)\) representing occurrences of \( b \)'s and \( a \)'s, respectively, are independent. However, we do not have \((A, a) I_{(A, A)} (b, A)\) whereas we do have \((A, a) I_{(b,a)} (b, A)\). It is only at state \((b, a)\) that the minimal requirements for \( I^M_q \) are met. Hence, \((b, A)\) and \((A, a)\) are concurrent at state \((b, a)\) but not at state \((A, A)\).
Figure 7.1: A well-behaved component, with two interfaces

It might also be worth noting that, with regard to Definition 7.1.2, \( u = (bb, aa) \) is an execution vector of the corresponding machine taking it from state \( q = (A, a) \) to state \( q' = (bb, aaa) \). In the notation of the definition, \((A, a) \rightarrow^{(bb, aa)} (bb, aaa)\).

The minimal requirement for concurrency at state \( q \in Q \) is depicted in Figure 7.2. Both independent transitions must be enabled at state \( q \), and both must occur between states \( q \) and \( q' \) in no particular order. It can be seen that \( I_q \) defines local concurrency in the sense that column vectors \( \xi_1, \xi_2 \) are concurrent at state \( q \) of the machine.

We may now refine \( \Sigma \)-machines to \( \Sigma \)-automata, taking into account both relations necessary to express concurrency as a structural property.
Definition 7.1.5. Let $\Sigma$ be a signature. A $\Sigma$-automaton $M$ is a $\Sigma$-machine $M = (Q, \succ)$ satisfying

1. If $\varepsilon _1 \overset{\Sigma}{\longrightarrow} _q q_1 \succ \varepsilon _2 \overset{\Sigma}{\longrightarrow} _q \tilde{q}$, then $q \succ \varepsilon _2 q_2 \succ \varepsilon _1 \tilde{q}$, some $q_2 \in Q$
2. If $q_1 \succ \varepsilon _1 \tilde{q}$ and $q_2 \succ \varepsilon _2 \tilde{q}$ and $q_1 \neq q_2$, then $\varepsilon _1$ and $\varepsilon _2$ and there exists $q \in Q$ such that $q \succ \varepsilon _2 q_1$ and $q \succ \varepsilon _1 q_2$.
3. If $q \in V\Sigma$ and $q \rightarrow^M q''$, then $\exists q' \in Q$ such that $q \rightarrow^M q' \Leftrightarrow q' \rightarrow^M q''$
4. If $\varepsilon _1, \varepsilon _2 \in E\Sigma$ s.t. $q \succ \varepsilon _1 \varepsilon _2$ and $\varepsilon \in V(M, q)$ with $\varepsilon _1, \varepsilon _2 \leq \varepsilon$, then $\varepsilon _1, \varepsilon _2 \overset{\Sigma}{\longrightarrow} _q q$

We also define a rooted $\Sigma$-automaton to be a rooted $\Sigma$-machine $M^* = (M, q)$ where $M$ is a $\Sigma$-automaton.

Note that by Definition 7.1.4 and (1) of Definition 7.1.5 we have that $I_q$ is symmetric and irreflexive. Symmetricity reflects the fact that concurrency is always mutual while irreflexivity prohibits considering an event as being concurrent with itself.

Condition (1) is characteristic of automata for non-interleaving representation of behaviour and is sometimes called the lozenge rule [Shi85, Shi97, NSW94]. Effectively, it says that if two independent events have occurred consecutively between states $q$ and $\tilde{q}$, then they have happened in no particular order. In other words, it should be possible for them to have occurred with their order interchanged. This is depicted in Figure 7.3.

![Figure 7.3: Condition (1) of Definition 7.1.5](image)

Condition (2) relates to discreteness of the generated language. A few words are in order to explain this further. Discreteness requires that elements bounded above in the
vector language have their least upper bound and greatest lower bound in it. As will be discussed in the following section (Section 7.2), this is the case when the generated language satisfies the lower diamond property (cf Definition 7.2.1). Informally, this property says that whenever we have the upper half of a diamond, then we have the whole diamond. The condition is illustrated in Figure 7.4. It can be seen that it results in lozenges in the graph of the automaton.

\[ \begin{array}{c}
\text{Figure 7.4: Condition (2) of Definition 7.1.5}
\end{array} \]

Condition (3) excludes the possibility that an execution vector may be produced in two different ways from sequences of individual transitions. In other words, when the first part of an execution vector takes us from its source state to an intermediate state, then the remaining part takes us from that state to the (execution vector's) target state. Dually, we may state the same for the second part of the execution vector. In fact, the only case that an execution vector can be produced in two different ways by sequences of individual transitions, is if the sequences differ only in the order of concurrent transitions. This is the point of condition (4). Condition (3) is depicted in Figure 7.5.

\[ \begin{array}{c}
\text{Figure 7.5: Condition (3) of Definition 7.1.5}
\end{array} \]
Condition (4) says that if two distinct transitions can start off the same behaviour from $q$, i.e. be part of the same execution vector from $q$, then they must do so concurrently. The motivation for this condition is not hard to see. Given a component vector $u$ which describes behaviour of the component at state $q$, the two distinct transitions $e_1, e_2$ essentially provide two different ways, say $u_1, u_2$, of extending to a behaviour described by $u$. In other words, $u - u_1 = x$ and $u - u_2 = x$. But this implies that $u_1$ and $u_2$ describe the same behaviour. Since, $e_1$ is a prefix of $u_1$, $e_2$ of $u_2$ and $e_1, e_2$ are distinct (and $e_1 \not\mid e_2 \land e_2 \not\mid e_1$ by condition (1) of $\Sigma$-machines), we can not have $u_1 = u_2$ in general. This will only be the case if $e_1, e_2$ are concurrent. This is the point of condition (4).

As a further note on Definition 7.1.5, it can be seen that condition (3) is a global rather than a local property. This makes checking against it difficult. [Shi05] establishes the following for this purpose. Let $M = (Q, \succ)$ be a $\Sigma$-machine and let $V \subseteq V_D$. If (a) there exists an onto function $\phi : V \rightarrow Q$ such that $\phi(u) \succ e \phi(u)$ iff $u = v \epsilon e$ and, (b) if $u, v \in V$ and $u \leq v$, then there exists $g \in E_D$ such that $u - g \in V$ and $u - g \leq v$, then $M$ satisfies condition (3) of Definition 7.1.5.

In further explanation of the move from $\Sigma$-machines to $\Sigma$-automata we note the following. In [SM04a] we started from a component $c$, constructed its corresponding discrete behavioural presentation, as described in Chapter 6, and from that extracted a transition system and an independence relation $\iota$ forming an ATS $C$, along the lines of the construction described in [Shi97]. Transitions of $C$ were labelled by $\lambda : A \rightarrow E_D$, where $A$ is an alphabet of actions of the system, and the independence relation was defined in the usual sense to give a concurrent alphabet $(E_D, \text{ind})$. The asynchronous transition system $C$ accepts a Mazurkiewicz trace language $\mathcal{L}(C, q)$ from any of its states $q$ and this language has an order structure corresponding to discreteness. The trace language $\mathcal{L}(C, q)$ can be related to $V_D$ by extending the function $\lambda$ to a monotonic function $\lambda^* : \mathcal{L}(C, q) \rightarrow V_D$. Thus, we get an asynchronous transition system together with a labelling function that extends to a mapping into a component language. This early version of $\Sigma$-automata can be found in [SM04a].

The drawback to this approach is that in general the function $\lambda^*$ is not injective, and
thus we can not induce discreteness of the component language generated by a \( \Sigma \)-automaton. In addressing this issue, subsequent analysis in [Shi05, MSK05] allowed for a significant part of the construction to be shortcut, mainly through the notion of \( \Sigma \)-machines, as described early in this section (see Definition 7.1.1). The main findings of this analysis suggested including conditions (2)-(4) of Definition 7.1.5 which were embedded in the structure of the corresponding \( \Sigma \)-automata.

This, perhaps, raises then the question as to whether such conditions are consistent with our intuitive requirements. If such requirements can be expressed in the context of a popular modelling language such as UML, then the answer is promising, as outlined in the following section.

7.1.1 UML state diagrams for \( \Sigma \)-automata

Execution vectors in our formal framework can be understood as compound transitions in UML. A compound transition in UML2.0 (see pp. 623-633 in [OMG04]) is used to represent a path of one or more transitions along the graph of a state diagram. It may include a \textit{join} which is used to merge several transitions emanating from states in different orthogonal regions or a \textit{fork} which is used to split an incoming transition to two or more transitions terminating on different orthogonal regions (see pp. 590-596 in [OMG04]). Orthogonal regions are different compartments of a composite state in UML state diagrams (see pp. 600-615 in [OMG04]) and they are used to represent concurrent states, by placing them in different regions.

Condition (3) of Definition 7.1.5 says that a compound transition can not be the result of different sequences of individual transitions. Recall that sequences which differ in the order of independent, consecutive events are not considered to be different in our formal framework, as discussed in Section 4.3.2.

Condition (2) says that if a compound transition starts off with a fork, in other words, if the head of a compound transition is a fork, then the fork leads to orthogonal regions. Further, condition (2) says that if the tail of a compound transition is a join, then the join is the result of exiting orthogonal regions.
7.1. Automata for Well-Behaved Components

Condition (4) is concerned with the formalisation of the case where the head of a compound transition is a fork targeting orthogonal regions of a composite state. Similarly, for the tail of a compound transition (apply condition (4) to the conclusion of condition (2)).

The only problem with establishing a straightforward relation between compound transitions in UML 2.0 and conditions (2)-(4) of $\Sigma$-automata in our formal framework, is that the semantics of compound transitions as given in the UML 2.0 specification document does not allow triggers on transitions entering a join or emanating from a fork (pp. 623-625 in [OMG04]). Note that this is irrespective of whether they are part of a compound transition or not.

The interpretation of UML compound transitions in our automata framework for components requires that transitions of a fork or a join can be labelled by an event, as done for example in the STATEMATE semantics of joins and forks (pp. 302-3 in [HN96]) in statecharts [Har87]. However, in this semantics the trigger of the fork or join is the conjunction of the triggers (events) of the individual transitions. This is achieved by simultaneity in our framework, which is modelled using column vectors with more than one non-empty coordinate. We shall see such a case in our example in Section 7.4. Here, we are concerned with a graphical representation of concurrent transitions since this would provide a state diagram-like notation for $\Sigma$-automata. Orthogonal regions are used in UML for concurrent states and thus this comes down to a formal interpretation of entering/exiting orthogonal regions in UML.

The available constructs in the standard UML notation for state diagrams [OMG04] suggest that it is appropriate to use a fork for entering the orthogonal regions followed by the use of a join for exiting them. This is depicted in the state diagram of Figure 7.6, where orthogonal regions, together with a fork and a join, are used to represent that $q_1, q_2$ are concurrent transitions.

The composite state $S2$ in the state diagram of Figure 7.6 has two orthogonal regions, one containing state $S3$ and the other $S4$, which can be visited concurrently. We make use of the fact that the head of a compound transition (in this case the compound transition involves $q_1, q_2$) may be a fork targeted to orthogonal regions (p.626 in [OMG04]).
Note that the transitions of the fork are entering the composite state $S_2$ explicitly, rather than by default, in UML dialect (p. 606 in [OMG04]).

Similarly, we make use of the fact that the tail of a compound transition in UML may be a join originating from orthogonal regions. Again, transitions of the join exit the composite state explicitly.

Therefore, we may represent diamonds in a component language, or lozenge shapes in a $Σ$-automaton, by a fork followed by a join in UML state diagrams. The semantics in this case is given in terms of the component language used in our formal framework. It says that either of the transitions of the fork construct can be taken and this determines the transition taken on the corresponding join construct. This reflects the fact that transitions appearing on a fork at state $q$ are related by $I_q$ in our formal framework and ensures that execution of the corresponding automaton goes round the lozenge of the underlying asynchronous transition system.
7.2 From Automata to Component Languages

In this section we describe how the vector language generated by a rooted $\Sigma$-automaton corresponds to the language of a well-behaved component. In other words, given a signature $\Sigma$, we want to ensure that a $\Sigma$-automaton generates a discrete and locally left-closed component language over $\Sigma$. This would allow designers to draw a state diagram for the usage protocol of a component and if this diagram adheres to the semantics prescribed earlier, it will correspond to a well-behaved component.

We have seen (Definition 7.1.2) that the vector language of a $\Sigma$-automaton comprises component vectors formed over $\Sigma$, and these are built up by (a series of) concatenations with column vectors from $E_\Sigma$ that appear on transitions of the automaton. Therefore, the execution vectors of a $\Sigma$-automaton decompose into products of column vectors. This can be exploited in showing that the vector language of a rooted $\Sigma$-automaton is locally left-closed.

Lemma 7.2.1. If $M = (Q; \succ)$ is a $\Sigma$-automaton and $q \in Q$, then $V(M, q)$ is locally left-closed.

Proof.
Let $u \in V(M, q)$ and $i \in I_\Sigma$ and suppose that $A < u \leq y(p)$. Then, we have column vectors $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n \in E_\Sigma$ such that $u = \varepsilon_1 \cdots \varepsilon_n$ and $q \succ \varepsilon_1 q_1 \cdots q_{n-1} \succ \varepsilon_n q_n$. Let $v_r = \varepsilon_1 \cdots \varepsilon_r$, then $v_r \in V(M, q)$, for all $r$, and $A \leq v_1(i) \leq \cdots \leq v_n(i) = y(i)$.

Since $v_{j+1}(i) = v_j(i)$ or $v_{j+1}(i) = v_j(i).e$, $e \in Op_\Sigma$, then $v_j(i) = u$, some $j$. Since $v_j \leq y$, by transitivity, there exists $v_j(i) \leq y(i)$, some $j$, and $v_j(i) = u$, which means that $V(M, q)$ is locally left-closed by Definition 4.2.3. □

Next we turn our attention to discreteness of the vector language generated by a rooted $\Sigma$-automaton, which is relatively more difficult to deal with. The analysis in [Shi05] shows that this is possible under the condition that whenever the structure of the automaton exhibits the upper half of a lozenge, then it exhibits the whole lozenge. In what follows we outline this development.

Recall (Definition 4.2.2) that discreteness requires that elements bounded above in a component language have their least upper bound and greatest lower bound in it.
We have seen that the order structure of a discrete component language exhibits the characteristic structure of a lattice. Therefore, discreteness of the generated language comes down to considering conditions under which partially ordered sets are (finite) lattices.

Suppose that \((X, \leq_X)\) is a partially ordered set. If \(x, y \in X\), then we define \(x \cup_X y\) (if it exists) to be the necessarily unique element of \(X\) which satisfies,

\[
\forall z \in X, x, y \leq_X z \iff x \cup_X y \leq_X z
\]

Dually, we may also define \(x \cap_X y\) to be the necessarily unique element of \(X\) which satisfies,

\[
\forall z \in X, x, y \geq_X z \iff x \cap_X y \geq_X z
\]

A subset \(U \subseteq X\) is a sublattice of \(X\) if \(x \cup_X y, x \cap_X y \in U\), for all \(x, y \in U\). Consequently, and by lifting these concepts to the partially ordered set of component vectors in our formal framework, we may say that \(V\) is a discrete subset of \(V_\Sigma\) precisely when \(\downarrow y = \{v \in V : y \leq u\}\) is a sublattice of \(V_\Sigma\), for all \(y \in V\). (Recall that \(V_\Sigma\) equipped with the ordering relation of Definition 4.1.1 (coordinate-wise prefix ordering) is a partially ordered set.)

In this way, the requirements for establishing discreteness have been narrowed down to considering conditions under which \(\downarrow y\), for all \(y \in V\), is a sublattice of \(V_\Sigma\). [Shi05] has shown that the required condition is a Church-Rosser type property which says that the presence of the upper half of a diamond infers the formation of the whole diamond. This 'downward proliferation of diamonds' can be formulated in terms of component vectors and languages as follows.

**Definition 7.2.1.** \(V\) will be said to have the lower diamond property (LDP) if and only if

\[
\forall u, v, w \in V, u \neq w \land u < w \land w < v \implies w = u \cup w \land u \cap v
\]

This property manifests itself in the structure of \(\Sigma\)-automata, in terms of states and transitions (see Definition 7.1.5(2)-(4)) and results in lozenge shapes in the graph of the automaton.
So far we have seen that $V$ is a discrete subset of $V_E$ when $\downarrow u = \{ v \in V : u \leq v \}$ is a sublattice of $V_E$, for all $u \in V$, and this is the case when $V$ has the LDP. Therefore, discreteness of the language comes down to proving that it has the LDP. This is done in the following lemma (the proof given here is an elaboration of the proof appearing in [Shi05]).

**Lemma 7.2.2.** If $M = (Q, \rightarrow)$ is a $\Sigma$-automaton and $q \in Q$, then $V(M, q)$ has the LDP.

**Proof.**
Suppose that $u, v, z \in V(M, q)$ such that $u \neq v$ and $u, v \triangleleft z$. We must show that $u \sqcup v = z$ and $u \sqcap v \in V(M, q)$.

By Proposition 4.2.1 it follows that $x/y = z \in E_\Sigma$ and $u \triangleleft z$. Hence, $u \triangleleft v \triangleleft z$. Therefore, there exists $q_u, \hat{q} \in Q$ such that $q \rightarrow u q_u$ and $q \rightarrow v \hat{q}$, where $\hat{q}$ is the unique state such that $q \rightarrow z \hat{q}$.

By Definition 7.1.5 (3), $q_u \triangleleft z \hat{q}$. If it is not the case that $q_u \triangleleft z \hat{q}$, then we may find $w$ with $A < w < z$ such that $q_u \rightarrow w q_u$ and $q \rightarrow w \hat{q} \hat{q}$, where $\hat{q}$ is the unique state such that $q \rightarrow z \hat{q}$. Hence, there exists $q_u$ such that $q \rightarrow u q_u \triangleleft z \hat{q}$.

Similarly, for $v$, there exists $z \hat{q}$ such that $z \hat{q} \triangleleft v$ and $q \rightarrow v \hat{q}$ such that $q \rightarrow u \hat{q} \triangleleft z \hat{q}$.

We consider two cases:

**Case 1:** $q_u \neq q_v$.

By Definition 7.1.5 (4), $e_u I_q e_v$ hence, $e_u \cap e_v = A$ and by Definition 7.1.5 (1), there exists $\hat{q} \in Q$ such that $\hat{q} \triangleleft q_u$ and $\hat{q} \triangleleft q_v$.

We first prove that $u \sqcup v = z$. We again have two cases:

1. $e_u(t) > A$, then $e_v(t) = A$ and we have $z(t) = u(t) \cdot e_u(t)$. Hence, $z(t) = \max(u(t), v(t))$.

2. $e_v(t) = A$, then $e_u(t) > A$, $u(t) \leq u(t) = e_v(t)$. Hence, $z(t) = \max(u(t), v(t))$.

Since $z(t) = \max(u(t), v(t))$, all $t \in I_\Sigma$, we have $z = u \sqcup v$. 

Now we prove that \( u \cap v \in V(M, q) \). Let \( u = u \cap v \). Since \( u, v \leq u \in V \Sigma \), by Proposition 4.1.2 we have

\[
u / (u \cap v) = (u \cup v) / u = u / v = e_u \]

It follows that \( u / w = e_u \), hence \( u = w \cdot e_u \). We now have \( q \rightarrow^{\Sigma \cdot u} q_u \) and \( q \rightarrow^{\Sigma} q_u \). By Definition 7.1.5 (3), \( q \rightarrow^{\Sigma} \bar{q} \). Hence, \( u \in V(M, q) \), which means that \( V(M, q) \) has the LDP in Case 1.

**Case 2:** \( q_u = q_v \). We have \( q_u \sim_{\Sigma} q_v \) and \( q_u = q_v \sim_{\Sigma} q_v \). By Definition 7.1.1 (1), \( e_u = e_v \). We also have, \( u \cdot e_u = v \cdot e_v \). It follows that \( u = v \), a contradiction. So, this case can never arise.

Thus, we have shown that \( V(M, q) \) has the LDP in any case. □

In summary, we have seen that the language \( V(M, q) \) generated by a rooted \( \Sigma \)-automaton is locally left-closed by Lemma 7.2.1, and discrete by Lemma 7.2.2 together with the reasoning prior to the definition of the LDP. Thus, it has been shown that \( (\Sigma, V(M, q)) \) is a well-behaved component.

Finally, it might be worth making a note of an interesting issue that comes to the fore if we consider the construction described in Section 7.1 for the well-behaved component \( (\Sigma, V(M, q)) = c \). From this component we may derive a \( \Sigma \)-automaton \( (M_c, A_\Sigma) \), following Definition 7.1.3, and this automaton in turn determines the same component, by Proposition 7.1.1. Hence, the same component \( (\Sigma, V(M, q)) \) is determined by the \( \Sigma \)-automaton \( (M, q) \), following the construction described in this section, and by the \( \Sigma \)-automaton \( (M_c, A_\Sigma) \), following the construction of Section 7.1.

The question arises as to the relationship between the automata \( (M, q) \) and \( (M_c, A_\Sigma) \). This will not be one of equality, in general. However, the relationship between them is a notion close to equality. The two \( \Sigma \)-automata may not be identical, but every action of one can be 'matched' by an action of the other. This is the essence of strong bisimulation in [Mil89]. This thinking is supported by the fact that, in the similar construction for the automata of [SM04a], two \( \Sigma \)-automata generate the same component language if and only if they are bisimilar (see Definition 10.1 and Proposition 10.3 in [SM04a]).
7.3 Composing $\Sigma$-automata

In this section, we give an account of composition of $\Sigma$-automata. The intention is to show that there is a way to put $\Sigma$-automata together and hence the automata-based formalism for components described in the previous sections is indeed compositional. A full account of composition within the automata-theoretic framework can be found in [Shi05].

In a fashion similar to that of composition of components, the key idea is that a component vector $\mathbf{u}$ represents behaviour of the product $M_1 || M_2$ providing it results from behaviours $\mathbf{v}_k$ of $M_k$, each $k$, which agree on coordinates corresponding to connected interfaces. The states of the composite automaton will be pairs of states, one from each of the constituent components, as usual. The more interesting part of composition has to do with the transition structure of the composite.

Before embarking on this, it should be made clear that in considering the composition of automata we find it convenient, albeit unpleasant, to deviate slightly from the notion of composition between component vectors. The deviation has to do with the common coordinate corresponding to the connected interface in the component vectors of the composite. In the case of component composition, this was internalised whereas in the case of automata we wish to keep the common coordinate. The reason is that the transition structure of the composite automaton is derived from the composition of the column vectors on transitions from each automaton. Activity on the common coordinate of the column vectors for the composite automaton indicates communication and thus change of state for both automata while activity on the rest of the coordinates indicates independent behaviour and incurs change of state for either automaton.

Keeping the common coordinate in the composite language comes down to removing the restriction to $I_{\Sigma_1} \Delta I_{\Sigma_2}$ in defining the operation of composition $\oplus$.

**Definition 7.3.1.** Let $c_1 = (\Sigma_1, V_1)$ and $c_2 = (\Sigma_2, V_2)$ be components and suppose that $\Sigma_1 \downarrow \Sigma_2$. We say that $\mathbf{u}_1 \in V_1$ and $\mathbf{u}_2 \in V_2$ are consistent, and we write $\mathbf{u}_1 \downarrow \mathbf{u}_2$, if

$$\mathbf{u}_1 |_{I_{\Sigma_1} \cap I_{\Sigma_2}} = \mathbf{u}_2 |_{I_{\Sigma_1} \cap I_{\Sigma_2}}$$
If \( u_1 \downarrow u_2 \), we define
\[
\| u_1 \| u_2 = (u_1 \cup u_2)
\]
where \( u_1 \cup u_2 : I_{\Sigma_1, \triangle} \) satisfies
\[
(u_1 \cup u_2)(i) = \begin{cases} 
  u_1(i), & i \in I_{\Sigma_1} \\
  u_2(i), & i \in I_{\Sigma_2}
\end{cases}
\]

It might be instructive to compare with Definition 5.1.4 of the \( \oplus \) operation.

The following definition provides notation for the 'projection' of vectors of the composite onto vectors of its constituent components. We shall find this notation useful in describing the transition relation of the composite automaton.

**Definition 7.3.2.** Suppose that \( \Sigma_1 \) and \( \Sigma_2 \) are signatures such that \( \Sigma_1 \downarrow \Sigma_2 \) and let vector \( \underline{v} \in V_1 \| V_2 \), then we define \( \underline{v}[k] = \underline{v}[I_{\Sigma_k}] \), where \( k = 1, 2 \).

This construction reminds of the projections which are used to define the trace semantics of parallel composition in COSY [JL92] and CSP [Hoa85], in the sense that if \( \underline{v} = u_1 \| u_2 \), then \( u_1 \) can be 'recovered' by restricting \( \underline{v} \) to \( I_{\Sigma_1} \), and likewise for \( u_2 \).

Notice that we use \( \| \) for composition in Definition 7.3.1 to avoid confusion with Definition 5.1.4. We will have more to say about this issue in the concluding chapter.

The basic idea behind the transition structure of the composite automaton is the following. The transition relation is given by \( (q_1, q_2) \rightarrow^\Sigma (q_1', q_2') \) if and only if for each \( k \) either \( \underline{e}[k] \neq A_{\Sigma_k} \) and \( q_k \rightarrow^{e[k]} q_k' \) or \( \underline{e}[k] = A_{\Sigma_k} \) and \( q_k = q_k' \). This is expressed more succinctly as \( q_k \rightarrow^{e[k]} q_k' \), each \( k \), in the following definition.

**Definition 7.3.3.** Let \( M = (Q, \rightarrow) \) be a \( \Sigma \)-machine and \( q_1, q_2 \in Q \), \( e \in E_\Sigma \), then we define
\[
q_1 \rightarrow^\Sigma q_2 \iff (q_1 \rightarrow^e q_2) \lor (e = A_\Sigma \land q_1 = q_2)
\]

Using \( q \) to denote that \( (q_1, q_2) \in Q_1 \times Q_2 \), and consequently \( q_k \) for \( q_k \), each \( k \), (in effect, \( q_k \) restricts \( q \) to \( q_1 \) or \( q_2 \), we may write \( q \rightarrow^\Sigma q' \) for the transition relation of the composite, which is translated in terms of the constituent automata as \( q_k \rightarrow^{e[k]} q_k' \), each \( k \).
In defining the transition relation of the composite we need to take into account two possibilities:

i) \( \varepsilon(i) \neq A \), \( i \in I_{\Sigma_1} \cap I_{\Sigma_2} \), in which case \( \varepsilon[i] \neq A_{\Sigma_k} \), each \( k \), and execution of the transitions from each automaton involves communication. This means that both transitions must be executed simultaneously so that the composite has a transition \((q_1, q_2) \xrightarrow{\varepsilon} (q'_1, q'_2)\).

ii) \( \varepsilon(i) = A \), all \( i \in I_{\Sigma_1} \cap I_{\Sigma_2} \), in which case there is no communication and execution of the transition of one of the constituent automata (the one for which \( \varepsilon[i] \neq A_{\Sigma_k} \), \( k = 1 \) or \( k = 2 \)) may occur independently of any transition in the other. Hence, the composite automaton has a transition \((q_1, q_2) \xrightarrow{\varepsilon} (q'_1, q'_2)\) if \( k = 1 \), and \((q_1, q_2) \xrightarrow{\varepsilon} (q_1, q'_2)\) if \( k = 2 \).

Note that while \( \varepsilon(i) = A \), all \( i \in I_{\Sigma_1} \cap I_{\Sigma_2} \), it is still possible for \( \varepsilon[i] \neq A_{\Sigma_k} \), each \( k \). For example, consider the column vectors \( e_1 = (\Lambda, b, \Lambda) \) and \( e_2 = (\Lambda, \Lambda, d, \Lambda) \), and suppose that the coordinate corresponding to the connected interface is the last coordinate of \( e_1 \) and the first of \( e_2 \). The coordinate corresponding to the connected interface in each, is empty. The two vectors are consistent, i.e. \( e_1 \perp e_2 \), and thus can be composed. Their composition gives \( e = e_1 \parallel e_2 = (\Lambda, b, \Lambda, \Lambda, d, \Lambda) \). We note that \( e[1] = (\Lambda, b, \Lambda) \neq A_{\Sigma_1} \) and \( e[2] = (\Lambda, \Lambda, d, \Lambda) \neq A_{\Sigma_2} \). In the resulting \( e \) the events \( b \) and \( d \) are simultaneous though this was clearly not the intention prior to composition.

It transpires that in cases where \( \varepsilon(i) = A \), for all \( i \in I_{\Sigma_1} \cap I_{\Sigma_2} \), and \( \varepsilon[i] \neq A_{\Sigma_k} \), each \( k \), composition would force the rest of the events (those appearing on the coordinates which correspond to the non-connected interfaces of each) to be simultaneous while this clearly need not be the case in general. To exclude such unfortunate situations, an additional requirement on the transition structure is that if the \( \varepsilon[i] \) are both non-null, then they must have some non-empty coordinate in common.

The above concepts are brought together in the formal definition of composition.

**Definition 7.3.4.** Suppose that \( M_1 = (Q_1, \xrightarrow{\cdot}) \) and \( M_2 = (Q_2, \xrightarrow{\cdot}) \) are \( \Sigma_k \)-machines, \( k = 1, 2 \) and \( \Sigma_1 \perp \Sigma_2 \) with \( \Sigma = \Sigma_1 \oplus \Sigma_2 \). We define \( M_1 || M_2 = (Q_1 \times Q_2, \xrightarrow{\cdot}) \), where \( \xrightarrow{\cdot} \subseteq (Q_1 \times Q_2) \times E_{\Sigma} \times (Q_1 \times Q_2) \) is given by \( q \xrightarrow{\varepsilon} q' \iff \)

1. \( q_k \xrightarrow{\varepsilon[i]} q'_k \), each \( k \)
2. If \( g(i) = \lambda \), all \( i \in I_{\Sigma_1} \cap I_{\Sigma_2} \), then either \( e[1] = \Delta_{\Sigma_1} \) or \( e[2] = \Delta_{\Sigma_2} \).

Point (1) of the definition gives the translation of the composite transition relation into those of the constituents and includes cases i) and ii) discussed above. Point (2) expresses the additional requirement that composition does not force otherwise independent column vectors to be necessarily simultaneous. Note that we cannot have \( e[1] = \Delta_{\Sigma_1} \) and \( e[2] = \Delta_{\Sigma_2} \) as this would give \( g = \Delta_{\Sigma} \) which contradicts the definition of transitions as column vectors in \( E_{\Sigma} \).

In order to show that our automata-theoretic framework is compositional, we need to establish that the composition (following Definition 7.3.4) of \( \Sigma \)-automata is itself a \( \Sigma \)-automaton. This involves a notion of compatibility between automata and a relationship between concurrency in the constituents and that of the composite automaton.

With regard to the discussion prior to Definition 7.3.4, we are concerned with the case where communication is involved. In this case, the corresponding components have (at least one) connected interfaces. The following definition says that the non-empty coordinates of the respective component vectors must agree.

**Definition 7.3.5.** Suppose that \( \Sigma_1 \) and \( \Sigma_2 \) are signatures with \( \Sigma_1 \downarrow \Sigma_2 \) and \( v_1 \in V_1 \), \( v_2 \in V_2 \), then we define

\[
v_1 \downarrow v_2 \iff \forall i \in I_{\Sigma_1} \cap I_{\Sigma_2} : v_1(i) = v_2(i)
\]

Also, we may define the set of events (labels on transitions) of a machine as follows.

**Definition 7.3.6.** If \( M = (Q, \succ) \) is a \( \Sigma \)-machine, then define

\[
E(M) = \{ e \in E_{\Sigma} \mid \exists q \in Q : q \succ^e \}
\]

In the case of column vectors, Definition 7.3.5 says that \( v_1 \downarrow v_2 \) if and only if on all non-empty common coordinates (which correspond to connected interfaces) \( v_1 \) and \( v_2 \) agree. Hence, \( \downarrow \) does not cater for cases where, say, \( v_1(i) = \lambda \land v_2(i) \neq \lambda \), \( i \in I_{\Sigma_1} \cap I_{\Sigma_2} \). The following definition rectifies this by imposing that if two transitions have at least one non-empty common coordinate on which they agree, then this must be the case for all their common coordinates.
**Definition 7.3.7.** Suppose that $\Sigma_1$, $\Sigma_2$ are signatures with $\Sigma_1 \downarrow \Sigma_2$ and $M_1$ and $M_2$ are $\Sigma_k$-machines, each $k$. We define $M_1 \downarrow M_2$ to be compatible, and we write $M_1 \downarrow M_2$, if

$$\forall e_1 \in E(M_1), \forall e_2 \in E(M_2). e_1 \downarrow e_2 \implies e_1 \downarrow e_2$$

This gives the compatibility condition within our automata-theoretic framework. One important consequence of this is that the execution vectors of the composite automaton are precisely those which project on execution vectors of the constituents, i.e. $(q_1, q_2) \rightarrow (q'_1, q'_2) \iff q_k \rightarrow (q'_k, q'_k)$ for each $k$.

We have seen that the move from $\Sigma$-machines to $\Sigma$-automata included considering local concurrency and this was expressed as a structural property of the resulting $\Sigma$-automaton. Therefore, before it can be shown that the composite of $\Sigma$-automata is a $\Sigma$-automaton, we need to establish a relationship between concurrency in the constituents and that of the composite automaton. Key to the treatment of local concurrency in a $\Sigma$-automaton was the independence relation (recall Definition 4.3.1). We address this first.

It is relatively straightforward to show that behaviours $u$ and $v$ of the composite automaton can take place independently if and only if their projections onto the constituent automata are independent.

**Remark 7.3.1.** Suppose that $c_1 = (\Sigma_1, V_1), c_2 = (\Sigma_2, V_2)$ are components with $\Sigma_1 \downarrow \Sigma_2$ and $c = c_1 \oplus c_2 = (\Sigma, V)$. Then, for all $u, v \in V$,

$$u \text{ ind } v \iff u[k] \text{ ind } v[k], \text{ each } k$$

**Proof.**

Suppose that $u \text{ ind } v$. We will show that $u[k] \text{ ind } v[k]$. Let $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$, then $u[k](i) > A$ which implies that $u(i) > A$. Since $u \text{ ind } v$ we have $u(i) = A$ which implies that $v[k](i) = A$. Hence, we have $u[k](i) > A$ and $v[k](i) = A$, all $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$, which gives $u[k] \text{ ind } v[k]$.

Conversely, suppose that $u[k] \text{ ind } v[k]$ and let $i \in I_{\Sigma}$. Then, $i \in I_{\Sigma_1}$, some $k$, and we have that $u(i) > A$ which implies that $u[k](i) > A$. Since $u[k] \text{ ind } v[k]$, all $i$, we have
\[ u(i) = \lambda \] which implies that \( y(i) = \lambda \). Hence, \( u(i) > \lambda \) and \( y(i) = \lambda \), all \( i \), which gives \( u \text{ ind } y \). □

Now we are in a position to define local potential concurrency in the constituent automata in terms of the translation of the composite transition relation for the constituents (Definition 7.3.3).

**Definition 7.3.8.** Suppose that \( M \) is a \( \Sigma \)-machine and \( q \in Q \) and \( \xi_1, \xi_2 \in E_{\Sigma} \). Then, we define

\[ \xi_1 I_{\xi_2}^M \iff \xi_1 \text{ ind } \xi_2 \wedge (\exists q', q'', \hat{q} \in Q : q \rightarrow^{\xi_1} q' \wedge q \rightarrow^{\xi_2} q'' \rightarrow^{\xi_1} \hat{q}) \]

The following lemma establishes the relationship between local potential concurrency in the constituent automata and local potential concurrency in the composite automaton.

**Lemma 7.3.1.** Suppose that \( M_1, M_2 \) are \( \Sigma_k \)-machines, \( k = 1, 2 \), and \( \Sigma_1 \perp \Sigma_2 \) with \( \Sigma = \Sigma_1 \oplus \Sigma_2 \) and \( M = M_1 \parallel | M_2 \). Then, for all \( \xi \in Q_1 \times Q_2 \) and \( c, \bar{f} \in B_{\Sigma} \)

\[ \xi I_{\bar{f}}^M \iff \xi I_{\bar{f}}^{M_k} \text{, each } k \]

*Proof.*

Suppose that \( \xi I_{\bar{f}}^M \), then by definition \( \xi \text{ ind } \bar{f} \) and there exists \( q', q'', \hat{q} \in Q_1 \times Q_2 \) such that \( q \rightarrow^\xi q' \) and \( q \rightarrow^\xi q'' \rightarrow^\xi \hat{q} \). So, for each \( k, \xi I_{\bar{f}}^{M_k} \) by Remark 7.3.1, and \( q \rightarrow^\xi q' \) and \( q \rightarrow^\xi q'' \rightarrow^\xi \hat{q} \). Hence, by Definition 7.3.8, \( \xi I_{\bar{f}}^{M_k} \) each \( k \).

Conversely, suppose that \( \xi I_{\bar{f}}^{M_k} \), each \( k \). Then, \( \xi I_{\bar{f}}^{M_k} \) by Definition 7.3.8, and there exist \( q_k', q_k'', \hat{q}_k \in Q_k \) such that \( q_k \rightarrow^{\xi_k} q_k' \) and \( q_k \rightarrow^{\xi_k} q_k'' \rightarrow^{\xi_k} \hat{q}_k \). Thus, \( \xi \text{ ind } \bar{f} \), by Remark 7.3.1, and \( q \rightarrow^\xi (q_1', q_2') \) and \( q \rightarrow^\xi (q_1'', q_2'') \rightarrow^\xi (\hat{q}_1, \hat{q}_2) \).

Since \( \xi \neq \Delta_{\Sigma} \neq \bar{f} \), we have that \( q \rightarrow^\xi (q_1', q_2') \) and \( q \rightarrow^\xi (q_1'', q_2'') \rightarrow^\xi (\hat{q}_1, \hat{q}_2) \) so that, by Definition 7.1.4, \( \xi I_{\bar{f}}^M \) as required. □

Having considered a notion of compatibility among transitions of the constituent automata and having related concurrency in the constituents to that of the composite, it can now be shown that the composite of \( \Sigma \)-automata is itself a \( \Sigma \)-automaton.
7.4 Illustration by example

In this section we apply the formal construction for associating a component with an automaton to obtain a state-oriented description of the CTuner component of the case study considered in Sections 4.4, 5.4 and 6.4.

We have seen (Section 5.4) that a formal description of the CTuner component is given by $c_T = (\Sigma_T, V_T)$ where

- $\Sigma_T$ is given by $\Sigma_T = (P_T, R_T, \beta_T)$ where $R_T = \{ISpeaker, IScreen, IAntenna\}$, $P_T = \{IDetectSignal\}$ and $\beta_T(IDetectSignal) = \{c_1\}$, $\beta_T(IAntenna) = \{g_1\}$, $\beta_T(ISpeaker) = \{d_1, d_2\}$, and $\beta_T(IScreen) = \{f_1, f_2\}$.

- its component language $V_T$ is given by

$$V_T = \{(A, A, A, A), (c_1, A, A, A), (c_1, d_1, A, A), (c_1, A, f_1, A), (c_1, d_1, f_1, A), (c_1, d_1, f_1, f_1, g_1)\}$$

The component language $V_T$ is discrete and locally left-closed (this has been asserted in Section 5.4 and can be readily checked against Definition 4.2.2 and Definition 4.2.3) and thus the CTuner component is well-behaved.

We may identify component vectors that are related by the covers relation (Definition 4.2.5) as follows.

$(A, A, A, A) \prec (c_1, A, A, A)$

$(c_1, A, A, A) \prec (c_1, d_1, A, A)$

$(c_1, d_1, A, A) \prec (c_1, d_1, f_1, A)$

$(c_1, A, f_1, A) \prec (c_1, d_1, f_1, A)$

$(c_1, d_1, f_1, A) \prec (c_1, d_1, f_1, g_1)$

$(c_1, d_1, f_1, g_1) \prec (c_1, d_1, f_1, f_2, g_1)$

It might be worth at this point to pause and make the observation that $(c_1, d_1, f_1, A)$ is not a primal vector in $V_T$. Referring back to the material presented in Chapter
6 (Section 6.4), it would not be associated with an event occurrence. This vector describes behaviour of the CTuner component in which $c1$ has happened and then $d1$ and $f1$ happened concurrently. It results from vectors $(c1, d1, A, A)$ and $(c1, A, f1, A)$, which are primal, and describe each of the concurrent event occurrences. We now return to associating the CTuner component with a $\Sigma$-automaton.

Since the CMenu component is well-behaved, the events associated with its interfaces are represented by the following column vectors (Proposition 4.2.1).

$$g_1 = (c1, A, A, A)$$

$$g_2 = (A, d1, A, A)$$

$$g_3 = (A, A, f1, A)$$

$$g_4 = (A, A, A, g1)$$

$$g_5 = (A, d2, f2, A)$$

We may now define a tuple $M_{\Sigma} = (V_T, \geq V_T)$ where $\geq V_T \subseteq V_T \times E_{V_T} \times V_T$ is defined by $u \geq v$ if and only if $\exists g \in V_T$ such that $u + g = v$ and $u/g = e$. Hence, the relation $\geq V_T$ between component vectors in $V_T$ is given as follows. We drop the subscript $V_T$ from now on.

$$(A, A, A, A) \geq_{g_1} (c1, A, A, A)$$

$$(c1, A, A, A) \geq_{g_2} (c1, d1, A, A)$$

$$(c1, A, A, A) \geq_{g_3} (c1, A, f1, A)$$

$$(c1, d1, A, A) \geq_{g_4} (c1, d1, f1, A)$$

$$(c1, A, f1, A) \geq_{g_5} (c1, d1, f1, A)$$

$$(c1, d1, f1, A) \geq_{g_4} (c1, d1, f1, g1)$$

$$(c1, d1, f1, g1) \geq_{g_6} (c1, d1d2, f1f2, g1)$$

We still need to show that $M_{\Sigma}$ is a $\Sigma$-machine. This is done by checking the conditions of Definition 7.1.1.

- By examination of the above relations, it can be seen that $u \geq v$ and $u \geq v'$ with $e \leq e'$ for no $u \in V_T$. Hence, condition (1) of the definition holds.
• By examination of the above relations, it can be seen that \( y \xrightarrow{e} y \) and \( y \xrightarrow{e'} y \)
with \( e \neq e' \) for no \( y, y \in V_T \). Hence, condition (2) of the definition holds.

Thus, \( M_{ep} = (V_T, \succ_{V_T}) \) is a \( \Sigma \)-machine.

In order to show further that \( M_{ep} \) is a \( \Sigma \)-automaton we need to consider local concurrency. We start by identifying independent column vectors in \( E_{\Sigma_T} \). We have,

\[
\begin{align*}
& e_1 \text{ ind } e_2, \quad e_4 \text{ ind } e_3, \quad e_1 \text{ ind } e_4, \quad e_1 \text{ ind } e_5, \\
& e_2 \text{ ind } e_3, \quad e_2 \text{ ind } e_4, \\
& e_3 \text{ ind } e_4.
\end{align*}
\]

According to Definition 7.1.4, the independent transitions \( e, e' \in E_{\Sigma_T} \) that realise the potential for concurrency are those that meet the additional requirement that there exist \( w_1, w_2, u \in V_T \) such that \( y \xrightarrow{e} w_1 \) and \( y \xrightarrow{e'} w_2 \xrightarrow{e} y \).

By examination of the \( \succ_{V_T} \) relation on component vectors in \( V_T \) given earlier, we conclude that this requirement applies only for \( e = e_2 \) and \( e' = e_3 \) since we have \( (c_1, A, A, A) \xrightarrow{e_2} (c_1, d_1, A, A) \) and \( (c_1, A, A, A) \xrightarrow{e_3} (c_1, A, f_1, A) \). 
(Take \( w_1 = (c_1, d_1, A, A), w_2 = (c_1, A, f_1, A) \) and \( y = (c_1, d_1, f_1, A) \) to see the exact match to the requirement.) Thus, \( e_2 I_{e_2} (c_1, A, A, A) e_3 \).

Having determined the \( \text{ind} \) and \( I_q \) relations on transitions of \( M_{ep} \), we examine whether it is a \( \Sigma \)-automaton by checking the conditions of Definition 7.1.5.

By examination of the \( \succ_{V_T} \) relation on component vectors in \( V_T \) given earlier, we conclude that condition (1) of the definition is relevant only for \( e_2, e_3 \in E_{\Sigma_T} \) and \( q = (c_1, A, A, A), q_1 = (c_1, d_1, A, A), q = (c_1, d_1, f_1, A) \in V_T \). In this case we have \( q \xrightarrow{e_3} q_2 \xrightarrow{e_3} q \), for \( q = (c_1, A, f_1, A) \in V_T \). Hence, condition (1) of Definition 7.1.5 holds.

By examination of the \( \succ_{V_T} \) relation on component vectors in \( V_T \) given earlier, we conclude that condition (2) of the definition is relevant only for \( e_2, e_3 \in E_{\Sigma_T} \) and \( q_1 = (c_1, d_1, A, A), q_2 = (c_1, A, f_1, A) \in V_T \). But for these column vectors we have that \( e_2 \text{ ind } e_3 \) and also \( q \xrightarrow{e_2} q_1 \) and \( q \xrightarrow{e_3} q_2 \) for \( q = (c_1, A, A, A) \in V_T \). Hence, condition (2) of Definition 7.1.5 holds.
For condition (3) we find it easier to check against the associated lemma. We define a function \( \phi : V_T \to Q_T \) by \( \phi(\mathbf{y}) = q_k, \ k = 0..6, \mathbf{y} \in V_T \) so that

\[
\begin{align*}
\phi((A, A, A, A)) &= q_0 \\
\phi((c_1, A, A, A)) &= q_1 \\
\phi((c_1, d_1, A, A)) &= q_2 \\
\phi((c_1, A, f_1, A)) &= q_3 \\
\phi((c_1, d_1, f_1, A)) &= q_4 \\
\phi((c_1, d_1, f_1, g_1)) &= q_5 \\
\phi((c_1, d_1 d_2, f_1 f_2, g_1)) &= q_6
\end{align*}
\]

By definition of \( \phi \) and the relation \( \succ_{V_T} \) on component vectors in \( V_T \) given earlier, condition (a) of the lemma holds. Condition (b) follows from the component language \( V_T \) and the 'q' relation between its component vectors given earlier. Hence, we may deduce that condition (3) of Definition 7.1.5 holds.

By examination of the \( \succ_{V_T} \) relation on component vectors in \( V_T \) given earlier, we conclude that condition (4) of the definition is relevant only for \( \varepsilon_2, \varepsilon_3 \in E_{2T} \) and \( q = (c_1, A, A, A), \mathbf{z} = (c_1, d_1, f_1, A) \in V_T \). But for these column vectors we have that \( \varepsilon_2 I_{(c_1, A, A, A)} \varepsilon_3 \) as explained in identifying local concurrency in \( V_T \) prior to checking the conditions. Hence, condition (4) of Definition 7.1.5 holds.

Thus, \( M_{c_T} = (V_T, \succ_{V_T}) \) is a \( \Sigma \)-automaton. It can be represented by the UML state diagram shown in Figure 7.7.

Placing \( (c_1, d_1, A, A) \) and \( (c_1, A, f_1, A) \) in different orthogonal regions of a composite state in the state diagram of Figure 7.7 indicates that the transitions from \( (c_1, A, A, A) \) to \( (c_1, d_1, f_1, A) \) take place in no particular order. This is reflected by the use of a fork emanating from \( (c_1, A, A, A) \), followed by a join targeting \( (c_1, d_1, f_1, A) \), for the concurrent transitions \( \varepsilon_2, \varepsilon_3 \). This is expressed in terms of the associated \( \Sigma \)-automaton by the fact that \( \varepsilon_2, \varepsilon_3 \) are related by \( I_2 \), i.e. \( \varepsilon_2 I_{(c_1, A, A, A)} \varepsilon_3 \).
7.5 Concluding note

In this chapter, our formal framework for components was extended with an automata-based formalism. Components were associated with \( \Sigma \)-automata which draw upon well-known concepts of ATS [Shi85, Bed88], but are particularly tailored to reflect properties of the underlying component language.

An extension to ATSs involves the so-called hybrid transition systems in which transitions are associated with multisets of event names (ch. 22 in [Shi97]). The \( \Sigma \)-automata described in this chapter lie somewhere between the two, in that the underlying asynchronous transition system is equipped with a specialised association of transitions to column vectors (in \( E_\Sigma \)), which may be interpreted as multisets.

The basic idea in associating a component with a \( \Sigma \)-automaton is to consider component vectors (which provide snapshots of behaviour (Chapter 3)) as states and define transitions in a way that captures the fact that component vectors are built up from...
the empty vector by repeatedly concatenating column vectors to it. The resulting transition system was then constrained by conditions necessary to express concurrency explicitly and ensure that the generated vector language corresponds to a well-behaved component. This led to the definition of a $\Sigma$-automaton for a well-behaved component.

This state-based description of component behaviour can be seen as the usage protocol state machine of the associated component. $\Sigma$-automata model both provides and requires assumptions and hence restrict the environment of a component in a fashion similar to the interface automata of [dHOlb]. In $\Sigma$-automata however, concurrency is represented as an explicit structural property of the automaton, and this is carried through to the structure of the composite automaton (Section 7.3). This allows to capture anomalies that may arise from the interplay of concurrency and nondeterminism, such as race conditions (Section 4.4), which can be tracked down to confusion in general net theory [Pet79b].

Furthermore, we considered the use of UML state diagrams for representing $\Sigma$-automata and argued that it is possible to represent concurrency in these diagrams using existing UML notation. This involved constructs for compound transitions, forks, joins and orthogonal regions. The only variation from the UML specification document [OMG04] has to do with allowing labels on transitions of a fork and a join.

In this chapter, we have presented the fundamentals of an automata-theoretic framework for components. More research is needed and we identify some immediate issues that require further attention.

We have seen (Section 7.1) that if we start with a well-behaved component $c = (\Sigma, V)$, then we have constructions

$$c = (\Sigma, V) \rightarrow (M_c, A_\Sigma) \rightarrow (\Sigma, V(M_c, A_\Sigma))$$

By Proposition 7.1.1, we have that $(\Sigma, V(M_c, A_\Sigma)) = c$, so the component generating the $\Sigma$-automaton and the component determined by that $\Sigma$-automaton are identical.

We have also seen (Section 7.2) that a rooted $\Sigma$-automaton $(M, q)$ generates a well-behaved component. If we couple this with the above, we now have constructions

$$(M, q) \rightarrow (\Sigma, V(M, q)) \rightarrow (M_c, A_\Sigma)$$
Hence, \((M, q)\) and \((M_c, \Delta_\Sigma)\) generate the same well-behaved component. The study of the automata of [SM04a], which follow exactly the same constructions, offers useful insights as to the relationship between the two \(\Sigma\)-automata and suggests that they are bisimilar, in the sense of strong bisimulation in [Mil89].

The composition of \(\Sigma\)-automata, described in Section 7.3, allows two approaches to the composition of components in relation to automata: either generate the corresponding components and then compose or compose the automata and then generate a component from the composite automaton. This offers interesting perspectives with regard to the preservation of well-behavedness under composition of components. This is further discussed in the concluding chapter of the thesis.
Chapter 7. State-oriented Description of Component Behaviour
Chapter 8

Concluding Remarks

The work in this thesis presents a formal framework for the specification of components which supports rigorous analysis and reasoning about their interactions and their composition. The additional information on the observable behaviours of components can be exploited within pragmatic approaches to software engineering in uncovering inconsistencies of scenario-based specifications.

In this chapter, we have included some concluding remarks along with a more detailed summary of the results of the thesis. We also discuss possible directions for future work.

8.1 Summary

It should be recognised that the specification of component-based software requires additional behavioural information about component interfaces. This information is necessary to analyse and reason about the behaviour of the system and also know what to expect when individual components are placed in a different context.

This research work has been motivated by the observation that current approaches to component-based software engineering lack a sound theoretical basis to support the engineering task. 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 behaviour at component interfaces. Industrial specifications using mainstream software engineering practices often suffer from inconsistencies that are due to the difficulty of defining concurrent interactions, and especially determining their interplay with alternative interactions.

Current formal approaches to the specification and analysis of components are concerned with concurrency arising through composition. This serves the all important purpose of interconnecting provided and required interfaces from different components, but does not capture concurrency at the interfaces of a single component. In reactive systems, upon receiving an event on a provided interface, the component may have to respond by generating events concurrently on two of its required interfaces.

We considered a language-based representation of component behaviour in Chapter 3. Each interface is allocated a specific coordinate in the component vectors of the language. In this way, at each point during a period of activation, the corresponding component vector records the sequences of events that have occurred on all interfaces of the component. In other words, each component vector provides a 'snapshot' of component behaviour. The set of component vectors that describe the intended behaviour of a component comprise its component language and this was obtained from UML2.0 sequence diagrams. We addressed their basic features for describing sequential, parallel and alternative behaviour. Furthermore, we introduced a construct for describing simultaneous events in a sequence diagram.

This was possible because the study of the order theoretic properties of component languages in Chapter 4 shows this language-based model to be expressive enough to capture mutually exclusive, concurrent and simultaneous events occurring on different interfaces of the same component. Concurrency in component languages, just like in vector languages of [Shi97], builds on the notion of concurrency found in independence models [Maz88, Shi86], but independence is considered at the vector level rather than the event level. This makes it possible to express concurrency at the individual component level, and also consider sets of simultaneous events on top of concurrent events on distinct component interfaces.

We required discreteness (a property related to finiteness and concurrency) and local
left-closure (related to ensuring every event occurrence is recorded in a component vector) of component languages to guarantee well-behavedness of the corresponding component. The order structure of component languages is dependent on context - on what other vectors are in the language. This means that missing behaviours identified in checking against discreteness and local left-closure have an effect on the orderings between events on component interfaces. This 'additional' behaviour often was not intended, in the sense that it only arises as a result of inconsistencies such as implicit scenarios due to race conditions. These in turn are the result of an interplay between concurrency and nondeterminism, which appears in situations like asymmetric confusion in General Net Theory [Pet79b].

Next, attention was confined to the composition of components. We identified conditions, in terms of signatures and languages, that determine component behaviours eligible for composition. The assembly and delegation connectors in UML2.0 were discussed in this respect. Components can then be composed via matching provided and required interfaces from each. This notion of composition is reminiscent of the parallel composition between processes in process algebras, where the process descriptions can be understood as the respective sequences of events from each interface. The composition of well-behaved components was shown to result in a well-behaved composite component, providing the individual components met a compatibility condition which takes up on the condition for composing component languages.

Apart from identifying inconsistencies in scenario-based specifications, the discreteness and local left-closure properties of component languages 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 transition systems. We were concerned with these connections in Chapters 6 and 7.

Discrete and locally left-closed component languages were associated with discrete behavioural presentations in Chapter 6. This builds on the notion of prime elements in posets which were interpreted as primal vectors in component languages. Using a behavioural presentation for a well-behaved component allows to use the temporal relations derived from this model for the orderings between event occurrences on com-
ponent interfaces. This means that it is appropriate to talk about true concurrency, just like in event structures, and also simultaneity (by considering the equivalence class generated by the pre-order and the mutual exclusion relations in behavioural 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.

Well-behaved components were associated with $\Sigma$-automata in Chapter 7. This association builds on consequences of discreteness and local left-closure of the corresponding component languages. $\Sigma$-automata are an elaboration of asynchronous transition systems [Shi85] and inherit the notion of concurrency expressed as consecutive, independent transitions that result in lozenge shapes. Their structure bears additional conditions that give discreteness and local left-closure of the generated component language. UML state diagrams can be used to represent $\Sigma$-automata and thus express concurrency explicitly, providing compound transitions are given a formal behavioural semantics in terms of our automata-based formalism. Composition of automata provides useful insights for component composition and these are further discussed in the following section.

Some of the results prior to writing up the thesis have been published in [MS03], [MSKF03], [MS04], [SM04], [Mos04], [SM04a], [MSK05], [MKS05].

Finally, we consider the following are valuable contributions of the thesis.

- A powerful formal framework for rigorous analysis and reasoning about components that incorporates a range of concurrency theories and considers concurrency at both the individual component level and the composition level. It is expressive enough to capture concurrency, nondeterminism and simultaneity, and allows for reasoning that reveals pathological behaviour arising through the interaction of such phenomena.

- The formal framework has been blended with mainstream software engineering practices, as exemplified by strong connections to UML. The formal underpinning it provides to UML diagrams concerned with specification aspects, allows to handle incomplete or inconsistent industrial specifications and provide feedback on pathological behaviours using the same specification technique (UML).
8.2 Future Directions

Many future developments of this work are possible. In this section we outline a number of the issues that arise.

The fact that we can move between a language-based and a state-oriented description of component behaviour within the same formal framework offers interesting perspectives with regard to moving swiftly between a scenario-based specification and a state-based specification. Moreover, this synthesis of behavioural specifications can be performed in a way that handles incomplete partial specifications that contain implicit assumptions, typically found in an industrial context.

The formal analysis and reasoning in identifying the complete set of intended behaviours should be supported by automated tools, and ideally be performed behind the scenes, while feedback is 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 other issues that arise from our work.

Aspects of nondeterminism

In our approach, we have used a component language to describe the behaviour of a component. We have seen (Chapter 4) that the relationships between event occurrences on different interfaces of a component are captured in the order structure of the corresponding component language. This allows to describe sequential, mutually exclusive, concurrent and simultaneous event occurrences on component interfaces.

However, there are limitations with respect to what can be expressed in this language-based description of a component. These have to do with nondeterminism between
event occurrences that refer to a single event and event occurrences that refer to a
(non-empty) simultaneity class of that event. Situations like this may be represented
by $\mathcal{L}_1, \mathcal{L}_2$ such that $\mathcal{L}_1 \leq \mathcal{L}_2$. It is not too difficult to see that this can be the case only
when the two column vectors agree on (at least) one non-empty coordinate but $\mathcal{L}_2$, say,
has (at least one) more non-empty coordinates than $\mathcal{L}_1$.

For instance, consider the column vectors $\mathcal{L}_1 = (a, A, A)$ and $\mathcal{L}_2 = (a, A, c)$. We note that
they are not independent. If they are both offered after some behaviour $\mathcal{L}_3 = (A, b, A)$,
then there is a choice between doing $a$ and $c$ simultaneously and $a$ on its own, as shown
in Figure 8.1.

![Figure 8.1: Event a either happens on its own or simultaneously with c](image)

The two occurrences (one is event $a$ on its own, the other is event $a$ simultaneously
with event $c$) are in conflict but this cannot be captured in the corresponding component language. Additional structure would be required to somehow ensure that
having done $a$ on its own (i.e. after $(a, b, A)$), it is not necessarily the case that $c$ occurs (i.e. $(a, b, A)$ does not stretch up to $(a, b, c)$). This may be seen more clearly if
we consider that the component language corresponding to Figure 8.1 would contain
$(A, b, A), (a, b, A), (a, b, c)$ and its order structure does not say more than "$b$ is followed
by $a$ which is followed by $c$".

Note that the nondeterministic choice between $a$ on its own and $a, c$ simultaneously
cannot be expressed irrespective of whether the intended behaviour is that $a$ on its own
8.2. Future Directions

can be followed by c. This would amount to a choice between either doing a and c in sequence or simultaneously. The situation is depicted in Figure 8.2.

\[
\begin{array}{c}
(a,b,c) \\
\downarrow \\
\vdots \\
(a,b, A) \\
\downarrow \\
\vdots \\
(A, b, A) \\
\downarrow \\
\vdots \\
\end{array}
\]

Figure 8.2: Events a and c either happen in sequence or simultaneously

It might be worth pointing out that irrespective of how the choice is resolved in this case, it results in a common behaviour \((a, b, c)\). This is not unsettling though, in the sense of 'forcing' concurrency, because the column vectors leading to it are not independent (and thus there is no potential for concurrency).

Note that situations like the ones described here, are excluded in \(\Sigma\)-automata (recall Definition 7.1.1(2) and also Definition 7.1.5(3)-(4)). These aspects of nondeterminism were beyond the scope of the present thesis and could be analysed in future work.

Future work should examine what additional structure is required in component languages in order to express these types of mutually exclusive event occurrences within our formal framework for components. One possibility could be the adoption of refusal sets used in the CSP trace model [BHR84] as a means of enforcing that certain events are not offered after certain events, even when the ordering between the corresponding component vectors would allow it. This is a subject for further investigation.

Obtaining component languages

The subset of features of sequence diagrams treated in obtaining the component language does not include the loop interaction fragment which is a construct for modelling
iteration. A loop [OMG04] is used in sequence diagrams to capture behaviour that is repeatedly executed while a certain condition holds. Its relevance to describing component interactions as such may be arguable. Nevertheless, future work could analyse an extension to the formal construction for obtaining a component language from a sequence diagram to consider the loop construct.

The issue of addressing a loop fragment at an interaction level can be seen in connection to addressing iteration of interactions (or scenarios) at the IOD (or hMSC) level. A promising solution lies with patterns of behaviour and further encouragement can be drawn by [SR02a] which shows that it suffices to consider a loop twice, once for entering the loop (condition holds) and once for exiting the loop (condition no longer holds).

**Automation**

It became apparent through the various examples and case studies that the process of translating a component’s participation in a sequence diagram into a component language is a tedious task in need of automation. Its highly repetitive nature makes this more appealing and future work should make this forthcoming.

Automation of the process can be seen in two phases. One is to assume that the sequence diagram is given (i.e. the set of locations and what each corresponds to whether it is a send/receive event, the start/end of an interaction fragment and so on). Then, automated support for the formal construction boils down to applying the appropriate definition (Definitions 3.2.2 - 3.2.6 from an algorithmic perspective) depending on the nature of the location, and this would require reasonable programming skills.

More advanced programming skills may be required for the second phase, which involves extracting the necessary information (along the lines of Definition 3.2.1) from a digital representation of a sequence diagram drawn using an automated tool. Then, the implementation of the first phase could be used as a back end. The effort (and graphics expertise) that goes into this second phase can be (re)used in generating sequence diagrams for describing instances of pathological behaviour, whilst checking for well-behavedness as exemplified in Chapter 4 (Section 4.4). Ideally, such feedback should
be passed on to the component designer(s) in the form of a sequence diagram.

Composition

In Chapter 5, we described a formal notion of composition of components (⊕). The idea is that if one component requires interface i and the other provides interface i, then a behaviour from one component and a behaviour from the other (as described by the respective component vectors) can only be composed if their restrictions to interface i are the same. From the composition of the behaviours the sequence of events corresponding to interface i is removed.

In Chapter 7, we associated components with automata and described a formal notion of composition of automata (Section 7.3). The transition structure of the automata comprises execution vectors and their composition (∥) follows the same principle underlying composition of components. The difference however is that in composing the execution vectors of automata, we keep the sequence of events corresponding to the connected interface, as discussed in Section 7.3.

The composition of automata allows two approaches to the composition of components in relation to automata: either generate the corresponding components and then compose or compose the automata and then generate a component from the composite automaton. This offers interesting perspectives with regard to the preservation of well-behavedness.

If the two approaches can be shown to commute, then we have for the language of the composite component that $V_1 \oplus V_2 = V(M_{c1}^* || M_{c2}^*)$ (the first approach is considered on the left side of the equation). This means that the composite automaton generates the language of the composite component, which is well-behaved (by Theorem 5.3.1). Thus, preservation of well-behavedness in automata composition would then come down to imposing conditions on $V_1, V_2$ such that the corresponding automata $M_{c1}^*, M_{c2}^*$ are compatible (Definition 7.3.5). This would, most likely, require a translation of the notion of compatibility between automata (∥) for the component languages.

Of course, this development presupposes a uniform notion of composition in terms of components (⊕) and corresponding automata (∥). The difference has to do with
keeping the common coordinate in the composite vectors. Adopting $||$ for composition of components seems more appealing from a component-based design perspective, since the connected provided interface may contain more operations than those used by the corresponding required interface (Lemma 5.1.1). These could be made available to some other component, subject to resolving the dependencies, if any, on operations already used in the existing connection. A related issue is identified in [RBF04] which is concerned with parametric component contracts. This is not possible if the connected interfaces are internalised by the resulting composite component.

On this evidence, adopting $||$ on component composition seems to be a more promising way forward. By comparing Definition 5.1.4 of $\oplus$ and Definition 7.3.1 of $||$, it can be seen that this boils down to removing the restriction to $I_{D_1} \triangle I_{D_2}$ in defining $\oplus$. This is relatively straightforward but has as a consequence that the composite signature also has to accommodate the connected interface. Preliminary analysis in [Shi05] suggests adding a set of local interfaces in the signature of a component (Definition 3.1.1).

**Definition 8.2.1.** We define a component signature to be a tuple $\Sigma = (P, L, R, \beta)$ where

- $P \subseteq I$ is a set of provided interfaces
- $L \subseteq I$ is a set of local interfaces
- $R \subseteq I$ is a set of required interfaces
- $\beta : P \cup L \cup R \to \varphi(Op)$; hence, $\beta(i)$ is the set of operations associated with interface $i$ of the component

and we require that $P \cap L \cap R = \emptyset$. Define $I_{D} = P \cup L \cup R$ and $Op_{D} = \bigcup_{i \in I_{D}} \beta(i)$.

The set $L$ in the definition would be empty initially, prior to composition, but could be used to keep the connected interfaces of the component after composition. In terms of the composite signature, the set of connected interfaces would comprise one interface for every matching pair (provided/required) of interfaces of the constituent components. The consequences that such a refinement to the theory might have on existing results on preservation of well-behavedness under $\oplus$ composition need to be further investigated.
8.2. Future Directions

Automata and Executable UML

The automata for describing component behaviour presented in Chapter 7 were related to UML state diagrams. This offered a way of modelling true concurrency in a state diagram. The executable subset of UML (xUML) comprises a version of state machines. There is no provision for concurrency however. It would be worthwhile to explore whether a version of the semantics of actions and events [WKC'03], which is the main driving force behind executable UML, can be extended to include a treatment of concurrency and subsequently be adopted within the transition structure of our automata-based formalism.

Infinite behaviour

We have presented a formal approach to the specification and analysis of components, which uses a component language to represent behaviour and reason about component interactions. A component language comprises a set of vectors each of whose coordinates corresponds to an interface and contains a finite sequence of events that have occurred on that interface. We have used UML sequence diagrams to obtain component languages, driven by the fact that scenario-based specifications are the mainstay of industrial specifications. In this context, it makes sense to talk about component languages as describing patterns of behaviour, which correspond to scenarios the component participates in.

However, there are certain classes of properties which include fairness and liveness notions such as progress, livelock and termination, whose formal treatment requires consideration of infinite behaviours. For example, fairness and related notions usually require that a component makes progress infinitely often. A mathematical framework in which fairness notions are expressible as infinitary progress properties has been formulated in [Kwi89], using a non-interleaving model of concurrency to distinguish between fairness related to concurrency and fairness of choice. This work has concentrated on excluding those infinite behaviours that are not fair, given a fairness property, while all finite behaviours are considered fair.
Future work should investigate how such properties can be discussed within our formal framework for components. We will need to extend component languages to consider infinite sequences of events on interfaces, by allowing component vector coordinates to contain both finite and infinite sequences. We should find the theory on infinitary languages developed in [Kwi89] of great use in this venture.

Below we present some preliminary ideas that should make the extension to infinitary component languages possible. We start by recalling the basic properties of infinite sequences and related concepts from order theory, and then attempt their application to the formal description of a component in our approach.

By $A^*$ we denote the set of all finite sequences formed over the set $A$. By $A^\omega$ we denote the set of all infinite sequences formed over the set $A$. Let $A^\infty = A^* \cup A^\omega$ denote the set of all finite and infinite sequences formed over $A$. For $n \in \mathbb{N}^+, x \in A^\infty$, $x_n$ denotes the $n$-th element of the sequence $x$ if it exists, and $\Lambda$ otherwise, where $\mathbb{N}^+$ denotes the set of natural numbers excluding zero.

It is well known that concatenation on sequences is an operation that takes two sequences $x$ and $y$ and produces a sequence $x.y$ as a result, starting with sequence $x$ and continuing with sequence $y$. If the sequence $x$ is infinite, then its concatenation with a sequence $y$ yields the sequence $x$ again. Put formally,$$
 x \in A^\omega, y \in A^\infty \implies x.y = x \in A^\omega
$$If the sequence $x$ is finite, then its concatenation with a sequence $y$ yields the sequence $x.y$ which is infinite (providing that the sequence $y$ is infinite; otherwise we have concatenation of finite sequences, which yields a finite sequence $x.y$ as already considered in our approach). Put formally,$$
 x \in A^*, y \in A^\infty \implies x.y \in A^\infty
$$A prefix order may be defined on $A^\infty$ and we write $x \leq_\infty y$ for sequences $x, y \in A^\infty$ if $x$ is a prefix of $y$. Put formally,$$
 \forall x, y \in A^\infty : x \leq_\infty y \iff \forall n \in \mathbb{N}^+ : n \leq |x| \implies x_n = y_n
$$Note that for all $x, y \in A^\omega$ we have $x \leq_\infty y \implies x = y$ and hence only prefixes of finite sequences can be distinguished.
8.2. Future Directions

The set $A^\infty$ together with the prefix ordering relation $\leq_\infty$ forms a complete partial order (CPO). Also, since the set $A^*$ of finite sequences of $A^\infty$ is countable, it can be deduced that $(A^\infty, \leq_\infty)$ is a domain (see Proposition 3.1.1 in [Kwi89]). Recall that a poset $(D, \leq)$ is a CPO if and only if

(i) $D$ has a least element

(ii) if $X \subseteq D$ and $X$ is a directed set, then $X$ has a least upper bound in $D$.

A set $X \subseteq D$ is a directed set if and only if it is non-empty and $\forall x, y \in X, \exists z \in X$ such that $x \leq z$ and $y \leq z$. Also, $X \subseteq D$ is a totally ordered set if for every $x, y \in X$, either $x \leq y$ or $y \leq x$.

Thus, the set of finite and infinite sequences $A^\infty$ ordered by $\leq_\infty$ is complete, in the sense that every directed set $X \subseteq A^\infty$ has a least upper bound in $A^\infty$ and $A^\infty$ has a least element (the empty sequence $\Lambda$).

Least upper bounds of directed sets of finite sequences can be used to describe infinite sequences, as done for example in [Bro00] where the set of all (finite and infinite) streams is a CPO and least upper bounds of finite streams are used to approximate infinite streams. A subset $L$ of $A^\infty$ is referred to as infinitary language in [Kwi89] and it is prefix closed if its set of prefixes, denoted by $Pref(L)$ comprises the language, i.e. $Pref(L) = L$.

In what follows we outline how the above concepts can be applied in our formal description of a component to obtain an infinitary component language.

By $Op^\infty_\Sigma$ we denote the set of all infinite sequences formed over the set of operations associated with a component with signature $\Sigma$. Let $Op^\infty_\Sigma = Op^*_\Sigma \cup Op^{\omega}_\Sigma$. By $\beta(i)^\omega$ we denote the set of all infinite sequences formed over the set of operations $\beta(i)$ corresponding to interface $i$ of the component. Similarly, let $\beta(i)^\infty = \beta(i)^* \cup \beta(i)^\omega$.

**Definition 8.2.2.** Suppose that $\Sigma$ is a signature. Define $V^\infty_\Sigma$ to be the set of all functions $y : I_\Sigma \rightarrow Op^\infty_\Sigma$ such that, for each $i \in I_\Sigma$, we have $y(i) \in \beta(i)^\infty$. We shall refer to elements of $V^\infty_\Sigma$ as $\infty$-vectors.
Thus, $V^\infty_\Sigma$ is the set of all component vectors formed over a signature $\Sigma$ and these vectors contain both finite and infinite sequences. A subset $V^\infty$ of $V^\infty_\Sigma$ is an infinitary component language. The infinite part of $V^\infty \subseteq V^\infty_\Sigma$, denoted by $V^\infty_{inf}$, is given by
$$V^\infty_{inf} = V^\infty \cap V^\infty_{\Sigma},$$
while the finite part of $V^\infty$, denoted by $V^\infty_{fin}$, is given by
$$V^\infty_{fin} = V^\infty \cap V^\infty_{\Sigma}.$$

We may define concatenation on vectors in $V^\infty_\Sigma$ by concatenating the finite and infinite sequences appearing on the corresponding coordinates, based on the operation of concatenation in $A^\infty$ discussed above. This is formally put in the following definition which is an adoption of the coordinate-wise concatenation on component vectors in $V_\Sigma$ (Definition 4.1.1) for $V^\infty_\Sigma$.

**Definition 8.2.3.** Let $u, v$ be vectors in $V^\infty_\Sigma$. Define $u \cdot v$ to be the unique vector $\beta \in V^\infty_\Sigma$ such that $\beta(i) = u(i) \cdot v(i) \in \beta(i)^\infty$, for each $i \in I_\Sigma$.

We may also define a prefix ordering between vectors in $V^\infty_\Sigma$ and write $u \preceq v$ for co-vectors $u, v \in V^\infty_\Sigma$ if every coordinate $i$ of $u$ is a prefix of the corresponding coordinate of $v$, in the sense of prefixes in the definition of $\leq_\infty$. This is formally put in the following definition.

**Definition 8.2.4.** Let $u, v$ be vectors in $V^\infty_\Sigma$. Define $u \preceq v$ if and only if $u(i) \leq_\infty v(i)$, for each $i \in I_\Sigma$.

In this way, we have extended prefix ordering onto $V^\infty_\Sigma$. Based on the observation that the set of sequences appearing on a particular coordinate of all co-vectors in a directed subset of an infinitary component language is totally ordered with respect to $\leq_\infty$, we may argue that an infinitary component language forms a complete partially ordered set (CPO). This development is outlined below.

If $X \subseteq V^\infty_\Sigma$ is a directed set, then the set of all (finite and infinite) sequences appearing on a particular coordinate $i$ of the co-vectors in $X$, given by $X_i = \{u(i) : u \in X\}$, is totally ordered. So each of the $X_i$ has a least upper bound and it is given as follows.

- If $X_i = \{u_1(i), u_2(i), \ldots\}$ with $u_1(i) \leq_\infty u_2(i) \leq_\infty \ldots$, then $\cup X_i = u_n(i), n \in \mathbb{N}^+$, if $u_n(i) = u_{n+1}(i) = \ldots$, and $(\cup X_i)_k = (u_n(i))_k$, if $|u_n(i)|_k \geq n$, where $k \in \mathbb{N}^+$ denotes the $k$-th element of the $n$-th sequence in $X_i$. 
8.2. Future Directions

Thus, every directed subset \( X \subseteq V^\omega \) has a least upper bound in \( V^\omega \) given by \( \bigvee X = (\bigvee X_i)_i \), each \( i \in I \), which is the \( \omega \)-vector each of whose coordinates is the least upper bound of the set of sequences appearing on the corresponding coordinate of the rest of the \( \omega \)-vectors in \( X \). This shows that \( V^\omega \) together with prefix ordering \( \leq \) on \( \omega \)-vectors forms a CPO.

We would like to use this extension to our framework of components to formally define liveness properties. The question arises as to what constitutes an acceptable infinitary component language.

From what we have seen, if \( V \subseteq V_\Sigma \) (whether it is normal or not), then it defines \( \overline{V} \subseteq V^\omega \) by \( v \in \overline{V} \) if and only if \( v = \bigvee X \), where \( X \subseteq V \) is a directed set. \( V \mapsto \overline{V} \) can be shown to be a closure operation [DP90] in the standard sense of order theory or topology. We could therefore consider an acceptable infinitary component language to be \( \overline{V} \), where \( V \) is normal.

On the other hand, we could attempt to deal with normality directly. In our component model, events occur on a single component interface sequentially - concurrency is only possible on distinct interfaces. We want to ensure that in the infinitary component language, each occurrence of an event on a particular interface will be described by distinct vectors. We thus require that an infinitary component language is locally left-closed, just like a component language in Section 4.2.

**Definition 8.2.5.** An infinitary component language \( V^\omega \) is locally left-closed if and only if, whenever \( u, v, w \in V^\omega \) and \( i \in I_\Sigma \), and \( x \in \beta(i)^+ \) such that \( x < u(i) \), then there exists \( u \in V^\omega \) such that \( v \leq u \) and \( u(i) = x \).

Note that we have required the sequence \( x \) to be finite in formulating the local left-closure property. This is to reflect that \( x \) has to be a prefix of \( u(i) \) and since only finite prefixes can be distinguished, we opted for considering \( x \) as being finite. The case that it is infinite can be included by simply taking \( x < u(i) \) in which case it will be equal to \( u(i) \).

It may be shown that if \( u, v, w \in V^\omega \), then \( u, v \leq w \Rightarrow u \cap w, u \cup w \in V^\omega \). So we could then define normality directly by analogy with the finite case already considered in our formal framework. We could conjecture that:
Chapter 8. Concluding Remarks

(a) If $V \subseteq V_{\Sigma}^\infty$ is normal, then $\overline{V}$ is normal
(b) If $V_{\Sigma}^\infty \subseteq V_{\Sigma}^\infty$ is normal, then $V_{\text{fin}} = V^\infty \cap V_{\Sigma}$ is normal
(c) $\overline{V} \cap V_{\Sigma} = V^\infty$

This would mean that normal infinitary component languages can be accepted by $\Sigma$-automata, described in Chapter 7, without major changes such as introducing states that are visited infinitely often.

Finally, it is worth noting that UML sequence diagrams which we use for obtaining a component language cannot express properties such as fairness or liveness. The use of liveness constraints in sequence diagrams has been proposed in [CKF04] in the form of the OCL template for liveness given in [BKS02]. This expresses necessity, interaction patterns that must occur, in terms of a progress becomes possible clause followed by a progress has been made clause. Such properties can be expressed in the distributed temporal true-concurrent logic MDTL [KF00b, KFO0a] which is interpreted over event structures. We have seen that component languages give rise to behavioural presentations which are related to event structures as discussed in Chapter 6.

A version of MDTL could be used as a specification language on top of component languages in place of or, even better, in combination with, sequence diagrams for enriched expressiveness. This is a worthwhile subject for investigation, especially considering that the combination of MDTL and OCL-enriched sequence diagrams could be used to express properties such as fairness and liveness of components which could then be translated into infinitary component languages.

8.3 Afterword

This work is one step closer to a form of "interactive design" which can be seen as a game between the tool supporting the formal reasoning (which tries to identify pathological behaviour) and the designer (who tries to prevent this). In this view of software engineering, practitioners can focus on development tasks best performed by people while the formal framework and accompanying tools can aid them in exploring the
consequences of design decisions on the set of intended behaviours of the components in question.

Formal methods are often frowned upon by practitioners in industry. Apart from accompanying 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 formally-based tool, produce high-quality specifications that provide compelling evidence that the behaviour of the system shall be predictable at all times.
Appendix A

Nested locations in UML2.0
sequence diagrams: an example

In Section 3.2.7, we described the inside-out approach to obtaining the component vectors associated with nested locations in a UML2.0 sequence diagram. This was demonstrated in Example 3.2.5 for a diagram comprising an alt interaction fragment within an operand of another alt fragment. Here, we demonstrate the inside-out approach for a mixture of interaction fragments, in particular we consider an alt fragment appearing within an operand of a par fragment. This also involves consideration of sequential, parallel and alternative locations from Section 3.2.

Consider a component $B$ whose intended behaviour within the context of a given scenario is described in the UML 2.0 sequence diagram of Figure A.1. Let $\Sigma_B$ be the signature of component $B$ given by $\Sigma_B = (P_B, R_B, \beta_B)$, where $P_B = \{i_1\}, R_B = \{i_2, i_3\}$ and $\beta_B(i_1) = \{a_1, a_2\}, \beta_B(i_2) = \{d_1, d_2, d_3\}$ and $\beta_B(i_3) = \{e_1, e_2\}$.

We write $(x, y, z)$ for component vectors in $V_{\Sigma_B}$ where $x = y(i_1)$, $y = y(i_2)$ and $z = y(i_3)$. In other words, the first coordinate of the component vectors corresponds to the interface $i_1$, the second to $i_2$ and the third to $i_3$. The set of all such $y$ formed over $\Sigma_B$ comprises the set of component vectors $V_{\Sigma_B}$. The formal description of the component $B$ is given as $c = (\Sigma_B, V_B)$ where its component language $V_B \subseteq V_{\Sigma_B}$ should comprise those component vectors only from $V_{\Sigma_B}$ that describe intended behaviour of
the component $B$.

Our objective is to obtain the component language $V_B$ from the scenario-based specification of component $B$ given in Figure A.1.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure.png}
\caption{Example of an alt fragment within a par fragment}
\end{figure}

$l_0$ is the initial location of the sequence diagram $npa$ and by Definition 3.2.2 it is mapped onto $A_{PB}$. Thus, $vec\_map(l_0) = (A, A, A)$.

The next location reached is location $l_1$ since $time(l_1) = time(l_0) + 1$. This location belongs to $Loc'$ and thus the associated component vectors are given by Definition 3.2.2 as follows.
vec_map(l_1) = \vec{u}_1 \text{ since } m = |vec_map(l_0)| = 1 \text{ and } time(l_0) = time(l_1) - 1. \text{ The component vector } \vec{u}_1 \text{ is given by }

\vec{u}_1 = (v_{i_1}, v_{i_2}, v_{i_3})

where each coordinate is obtained as,

- \(v_{i_1} = u_{i_1} \cdot a_1 = a_1 \text{ since } (a_1, l_1) \in RE_B \land a_1 \in \beta_B(i_1)\)
- \(v_{i_2} = u_{i_2} = \Lambda \text{ since } (a_1, l_1) \in RE_B \text{ but } a_1 \not\in \beta_B(i_2)\)
- \(v_{i_3} = u_{i_3} = \Lambda \text{ since } (a_1, l_1) \in RE_B \text{ but } a_1 \not\in \beta_B(i_3)\)

Thus, vec_map(l_1) = \vec{u}_1 = (v_{i_1}, v_{i_2}, v_{i_3}) = (a_1, \Lambda, \Lambda).

The next location reached is location \(l_2\) (since \(time(l_2) = time(l_1) + 1\)). This location marks the beginning of a \texttt{par} interaction fragment with 2 operands, since \(scope(l_2) = npa.par(2)\). Nevertheless, we still have that \(l_2 \in Loc\) and thus its associated component vectors can be obtained following the construction given in Definition 3.2.2 as follows.

\(vec_map(l_2) = \vec{u}_2 \text{ since } m = |vec_map(l_1)| = 1 \text{ and } time(l_1) = time(l_2) - 1. \text{ The component vector } \vec{u}_2 \text{ is given by }

\vec{u}_2 = (v_{i_2}, v_{i_2}, v_{i_2})

where each coordinate is obtained as,

- \(v_{i_2} = u_{i_2} = a_1 \text{ since no event from } \beta_B(i_1) \text{ is associated with } l_2\)
- \(v_{i_2} = u_{i_2} = \Lambda \text{ since no event from } \beta_B(i_2) \text{ is associated with } l_2\)
- \(v_{i_2} = u_{i_2} = \Lambda \text{ since no event from } \beta_B(i_3) \text{ is associated with } l_2\)

Thus, vec_map(l_2) = \vec{u}_2 = (v_{i_2}, v_{i_2}, v_{i_2}) = (a_1, \Lambda, \Lambda).

Notice that \(vec_map(l_2) = (a_1, \Lambda, \Lambda)\) which is precisely \(vec_map(l_1)\) since no event of the component is associated with location \(l_2\). This location is simply marking the beginning of the \texttt{par} fragment but we need to consider it as it has a role to play in
mapping the subsequent locations appearing within the \texttt{par} fragment onto component vectors.

The next location to be considered is $l_3$. It is not necessarily the next one to be reached, at least not on its own, as explained in Section 3.2.5. This location is the first location of the 1st operand of the \texttt{par} fragment since $\text{time}(l_3) = \text{time}(l_2) + 1 \land \text{scope}(l_3) = \text{npa.par}(2)\|$1 while $\text{scope}(l_2) = \text{npa.par}(2)$. This may be expressed more succinctly by saying that $l_3 \in Y_2$. Thus, its associated component vectors are given by Definition 3.2.5 as follows.

\[
\text{vec.map}(l_3) = \psi_2 \text{ since } m = |\text{vec.map}(l_2)| = 1 \text{ and } \text{scope}(l_2) = \text{npa.par}(2). \]

The component vector $\psi_3$ is given by

\[
\psi_3 = (\psi_{l_3}, \psi_{l_2}, \psi_{l_3})
\]

where each coordinate is obtained as,

\begin{itemize}
    \item $\psi_{l_3} = \psi_{l_2} = a_1$ since $(l_3, d_1) \in SE_B$ but $d_1 \notin \beta_B(l_1)$
    \item $\psi_{l_3} = \psi_{l_2} = d_1$ since $(l_3, d_1) \in SE_B \land d_1 \in \beta_B(l_2)$
    \item $\psi_{l_3} = \psi_{l_2} = A$ since $(l_3, d_1) \in SE_B$ but $d_1 \notin \beta_B(l_3)$
\end{itemize}

Thus, \text{vec.map}(l_3) = \psi_3 = (\psi_{l_3}, \psi_{l_2}, \psi_{l_3}) = (a_1, d_1, A).

The next location reached is $l_4$ and this is compulsory if $l_3$ was visited, since $\text{time}(l_3) = \text{time}(l_4) - 1$ and $l_3, l_4$ belong to the same operand. For this location we have $\text{scope}(l_4) = \text{npa.par}(2)\|$1.\texttt{alt}(2) and so it is in \texttt{par} but also marks the beginning of a nested \texttt{alt} fragment with 2 operands.

We apply the \textit{inside-out} approach described in Section 3.2.7, which says that we first consider what component vectors are obtained from the fact that $l_4$ is in \texttt{alt} (the contained fragment) and then consider what additional component vectors are associated with the location due to the fact it is also in \texttt{par} (the enclosing fragment).

We have that $l_4 \in Loc'$ and thus its associated component vectors are given by Definition 3.2.2 as follows.
\[
\text{vec.map}(l_4) = v_{l_4} \text{ since } m = |\text{vec.map}(l_4)| = 1 \text{ and } \text{time}(l_3) = \text{time}(l_4) - 1. \text{ The component vector } v_{l_4} \text{ is given by }
\]

\[
v_{l_4} = (v_{l_{4_1}}, v_{l_{4_2}}, v_{l_{4_3}})
\]

where each coordinate is obtained as,

- \(v_{l_{4_1}} = v_{l_{4_2}} = a_1\) since no event from \(\beta_B(i_1)\) is associated with \(l_4\)
- \(v_{l_{4_2}} = v_{l_{4_3}} = d_1\) since no event from \(\beta_B(i_2)\) is associated with \(l_4\)
- \(v_{l_{4_3}} = v_{l_{4_5}} = \Lambda\) since no event from \(\beta_B(i_3)\) is associated with \(l_4\)

Thus, \(\text{vec.map}(l_4) = v_{l_4} = (v_{l_{4_1}}, v_{l_{4_2}}, v_{l_{4_3}}) = (a_1, d_1, \Lambda)\).

Notice again that \(\text{vec.map}(l_4) = (a_1, d_1, \Lambda)\) which is precisely \(\text{vec.map}(l_3)\) since no event has happened in moving from location \(l_3\) to \(l_4\). Hence, location \(l_4\) is not associated with any event of the component. It is simply marking the beginning of the contained or nested \(\text{alt}\) and its role is important in determining the component vectors for the first location of each operand in the nested \(\text{alt}\) fragment.

The next location to be considered (not necessarily the next one to be reached, as location \(l_7\) may also be reached directly after \(l_4\)) is \(l_5\). This is the first location of the 1st operand of \(\text{alt}\), since \(\text{time}(l_5) = \text{time}(l_4) + 1 \wedge \text{scope}(l_5) = \text{npa.par}(2)\text{alt}(2)\text{alt}(2)\) while \(\text{scope}(l_4) = \text{npa.par}(2)\text{alt}(2)\text{alt}(2)\). This may be expressed more succinctly by saying that \(l_5 \in Y_1\). Consequently, its associated component vectors are given by Definition 3.2.3 as follows.

\[
\text{vec.map}(l_5) = v_{l_5} \text{ since } m = |\text{vec.map}(l_4)| = 1 \text{ and } \text{scope}(l_4) = \text{npa.par}(2)\text{alt}(2). \text{ The component vector } v_{l_5} \text{ is given by }
\]

\[
v_{l_5} = (v_{l_{5_1}}, v_{l_{5_2}}, v_{l_{5_3}})
\]

where each coordinate is obtained as,

- \(v_{l_{5_1}} = v_{l_{4_1}} = a_1\) since \((l_5, d_2) \in SE_B\) but \(d_2 \notin \beta_B(i_1)\)
- \(v_{l_{5_2}} = v_{l_{4_2}} = d_2\) since \((l_5, d_2) \in SE_B \wedge d_2 \in \beta_B(i_3)\)
\[ v_{i_3} = v_{l_4} = A \text{ since } (l_5, d2) \in SE_B \text{ but } d2 \not\in \beta_B(i_3) \]

Thus, \( vec.map(l_5) = \psi_5 = (v_{l_1}, v_{l_2}, v_{l_3}) = (a1, d1d2, A) \).

The next location reached is \( l_6 \) and this is compulsory if \( l_5 \) was visited, since \( time(l_5) = time(l_6) - 1 \) and \( l_5, l_6 \) belong to the same operand of \( \text{alt} \). This location belongs to \( Loc' \) and hence its component vectors are given by Definition 3.2.2 as follows.

\[ vec.map(l_6) = \psi_6 \text{ since } m = |vec.map(l_5)| = 1 \text{ and } time(l_6) = time(l_5) - 1. \]

The component vector \( \psi_6 \) is given by

\[ \psi_6 = (v_{l_1}, v_{l_2}, v_{l_3}) \]

where each coordinate is obtained as,

- \( v_{l_1} = v_{l_1} = a1 \text{ since } (l_6, e1) \in SE_B \text{ but } e1 \not\in \beta_B(i_1) \)
- \( v_{l_2} = v_{l_2} = d1d2 \text{ since } (l_6, e1) \in SE_B \text{ but } e1 \not\in \beta_B(i_2) \)
- \( v_{l_3} = v_{l_3} = e1 \text{ since } (l_6, e1) \in SE_B \text{ and } e1 \in \beta_B(i_3) \)

Thus, \( vec.map(l_6) = \psi_6 = (v_{l_1}, v_{l_2}, v_{l_3}) = (a1, d1d2, e1) \).

The next location to be considered is \( l_7 \) (which can be reached only directly from \( l_4 \)) This location is the first location the 2nd operand of the \( \text{alt} \) fragment, since we have \( time(l_7) = time(l_4) + 1 \text{ and scope}(l_7) = npa.par(2)\|1.\text{alt}(2)\|2 \) while \( scope(l_5) = npa.par(2)\|1.\text{alt}(2)\|1 \). This may be expressed more succinctly by saying that \( l_7 \in Y_1 \).

Consequently, its associated component vectors are given by Definition 3.2.3 as follows.

\[ vec.map(l_7) = \psi_7 \text{ since } m = |vec.map(l_4)| = 1 \text{ and } scope(l_4) = npa.par(2)\|1.\text{alt}(2). \]

The component vector \( \psi_7 \) is given by

\[ \psi_7 = (v_{l_1}, v_{l_2}, v_{l_3}) \]

where each coordinate is obtained as,

- \( v_{l_1} = v_{l_1} = a1 \text{ since } (l_7, e2) \in SE_B \text{ but } e2 \not\in \beta_B(i_1) \)
- \( v_{l_2} = v_{l_2} = d1 \text{ since } (l_7, e2) \in SE_B \text{ but } e2 \not\in \beta_B(i_2) \)
- \( v_{l_3} = v_{l_3} = e2 \text{ since } (l_7, e2) \in SE_B \text{ and } e2 \in \beta_B(i_3) \)
• \( v_{t_8} = v_{l_8}.e2 = e2 \) since \((l_7, e2) \in SE_B \land e2 \in \beta_B(i_3)\)

Thus, \(\text{vec.map}(l_7) = \chi_{l_7} = (v_{l_1}, v_{l_2}, v_{l_3}) = (a1, d1, e2)\).

The next location reached is \(l_8\) and this is compulsory if \(l_7\) was visited, since \(\text{time}(l_7) = \text{time}(l_8) - 1\) and \(l_7, l_8\) belong to the same operand of \(\text{alt}\). This location belongs to \(\text{Loc'}\) and hence its component vectors are given by Definition 3.2.2 as follows.

\[
\text{vec.map}(l_8) = \chi_{l_8} \quad \text{since} \quad m = |\text{vec.map}(l_7)| = 1 \quad \text{and} \quad \text{time}(l_7) = \text{time}(l_8) - 1.
\]

The component vector \(\chi_{l_8}\) is given by

\[
\chi_{l_8} = (v_{l_1}, v_{l_2}, v_{l_3})
\]

where each coordinate is obtained as,

• \( v_{l_1} = v_{l_1} = a1 \) since \((l_8, d3) \in SE_B\) but \(d3 \notin \beta_B(i_1)\)

• \( v_{l_2} = v_{l_2} = d1d3 \) since \((l_8, d3) \in SE_B \land d3 \in \beta_B(i_2)\)

• \( v_{l_3} = v_{l_3} = e2 \) since \((l_8, d3) \in SE_B\) but \(d3 \notin \beta_B(i_3)\)

Thus, \(\text{vec.map}(l_8) = \chi_{l_8} = (v_{l_1}, v_{l_2}, v_{l_3}) = (a1, d1d3, e2)\).

The next location considered is \(l_9\) (which is reached either directly after \(l_6\) was visited or directly after \(l_8\)). This location marks the end of the nested \(\text{alt}\) fragment, since \(\text{scope}(l_9) = \text{npa.par}(2)\|1.\text{alt}(2)\). Thus, the component vectors associated with \(l_9\) are given by Definition 3.2.4 as follows.

\[
\text{vec.map}(l_9) = \chi_{l_9}^{(1)} \cup \chi_{l_9}^{(2)} = \text{vec.map}(l_6) \cup \text{vec.map}(l_8)
\]

since \(l_9\) is the last location of the 1st operand of the nested \(\text{alt}\) (because \(\text{time}(l_7) = \text{time}(l_8) + 1\) \land \text{scope}(l_6) = \text{npa.par}(2)\|1.\text{alt}(2)\|1\) while \(\text{scope}(l_7) = \text{npa.par}(2)\|1.\text{alt}(2)\|2\)) and \(l_9\) is the last location of the 2nd operand of the nested \(\text{alt}\) (because \(\text{time}(l_7) = \text{time}(l_8) + 1\) \land \text{scope}(l_6) = \text{npa.par}(2)\|1.\text{alt}(2)\|2\) while \(\text{scope}(l_9) = \text{npa.par}(2)\|1.\text{alt}(2))\).

Thus, \(\text{vec.map}(l_9) = \{\chi_{l_9}^{(1)}, \chi_{l_9}^{(2)}\} = \{(a1, d1d2, e1), (a1, d1d3, e2)\}\).

Up to this point we have obtained the component vectors associated with \(l_0, l_1, l_2, l_3\) and have partly considered locations \(l_4 - l_9\). We say partly because we have only addressed
them as locations of the nested alt fragment. We still have to address locations $l_4 - l_9$ as belonging to the enclosing par fragment. Also, note that we have obtained the component vectors for locations $l_2, l_3$ but these locations will be considered again in computing cases I, II, III for obtaining the vectors of the end location $l_{11}$ of the par fragment in view of Definition 3.2.6.

The next location considered is location $l_{10}$. This location is the first location of the 2nd operand of par, since $\text{time}(l_{10}) = \text{time}(l_9) + 1 \land \text{scope}(l_{10}) = \text{npar.par}(2)$ while $\text{scope}(l_9) = \text{npar.par}(2) \& \& \text{alt}(2)$. This may be expressed more succinctly by saying that $l_{10} \in Y_2$.

$\text{ve.map}(l_{10}) = \psi_1$ since $m = |\text{ve.map}(l_2)| = 1$ and $\text{scope}(l_2) = \text{npar.par}(2)$. The component vector $\psi_{10}$ is given by

$$\psi_{10} = (\psi_{10_1}, \psi_{10_2}, \psi_{10_3})$$

where each coordinate is obtained as,

- $\psi_{10_1} = \psi_{l_{10_1}} = a_2 = a_1 a_2$ since $(a_2, l_{10}) \in \text{RE}_B \land a_2 \in \beta_B(i_1)$
- $\psi_{10_2} = \psi_{l_{10_2}} = \Lambda$ since $(a_2, l_{10}) \in \text{RE}_B$ but $a_2 \notin \beta_B(i_2)$
- $\psi_{10_3} = \psi_{l_{10_3}} = \Lambda$ since $(a_2, l_{10}) \in \text{RE}_B$ but $a_2 \notin \beta_B(i_3)$

Thus, $\text{ve.map}(l_{10}) = \psi_{10} = (\psi_{l_{10_1}}, \psi_{l_{10_2}}, \psi_{l_{10_3}}) = (a_1 a_2, \Lambda, \Lambda)$.

The next location considered is $l_{11}$. This is the end location of the par fragment since $\text{scope}(l_{11}) = \text{npar.par}(2)$. Thus, we apply the construction given in Section 3.2.5 and then Definition 3.2.6. This gives the component vectors associated with the location $l_{11}$ and this is what feeds into the resulting component language $V_B$ from the par fragment.

First, we must compute $\text{ve.map}(l)^I, \text{ve.map}(l)^II, \text{ve.map}(l)^III$ for each location $l = l_3, l_4, l_5, l_6, l_7, l_8, l_9, l_{10}$.

We start with $l_3$.

- $\text{ve.map}(l_3)^I = \text{ve.map}(l_3) = (a_1, d_1, \Lambda)$
- \( \text{vec.map}(l_3)^{II} = \text{vec.map}(l_{10}) = (a1a2, \Lambda, \Lambda) \)
  since \( l_{10} \) is the only location in the 2nd operand of \( \text{par} \)

- \( \text{vec.map}(l_3)^{III} = \psi_{10}, l_0 \)
  since \( l_{10} \) is the only location of the 2nd operand of \( \text{par} \). The corresponding component vector is given by
  \[
  \psi_{10} = \psi_3 \sqcup \psi_{10} = (a1, d1, \Lambda) \sqcup (a1a2, \Lambda, \Lambda) = (a1a2, d1, \Lambda)
  \]

The cases \( I, II, III \) for location \( l_4 \) are as follows.

- \( \text{vec.map}(l_4)^{I} = \text{vec.map}(l_4) = (a1, d1, \Lambda) \)

- \( \text{vec.map}(l_4)^{II} = \text{vec.map}(l_{10}) = (a1a2, \Lambda, \Lambda) \)
  since \( l_{10} \) is the only location in the 2nd operand of \( \text{par} \)

- \( \text{vec.map}(l_4)^{III} = \psi_{10}, l_{10} \)
  since \( l_{10} \) is the only location of the 2nd operand of \( \text{par} \). The corresponding component vector is given by
  \[
  \psi_{10} = \psi_4 \sqcup \psi_{10} = (a1, d1, \Lambda) \sqcup (a1a2, \Lambda, \Lambda) = (a1a2, d1, \Lambda)
  \]

It is worth pointing out that location \( l_4 \) does not add any new component vectors (as compared to those of \( l_3 \)) to the resulting component language, as we will see when this is obtained below, because it marks the beginning of the nested \texttt{alt} but it is not involved in any event occurrence as such. In effect, it 'inherits' the component vectors of its immediately preceding location. Its contribution lies with obtaining the component vectors associated with the nested \texttt{alt} fragment rather than the enclosing \( \text{par} \).

We continue with location \( l_5 \).

- \( \text{vec.map}(l_5)^{I} = \text{vec.map}(l_5) = (a1, d1d2, \Lambda) \)

- \( \text{vec.map}(l_5)^{II} = \text{vec.map}(l_{10}) = (a1a2, \Lambda, \Lambda) \)
  since \( l_{10} \) is the only location in the 2nd operand of \( \text{par} \)
• \( \text{vec.map}(l_5)^{III} = \mathcal{U}_{l_5,l_{10}} \)
  since \( l_{10} \) is the only location of the 2nd operand of \( \text{par} \). The corresponding component vector is given by
  \[
  \mathcal{U}_{l_5,l_{10}} = \mathcal{U}_0 \cup \mathcal{U}_{l_{10}} = (a_1, d_1 d_2, A) \sqcup (a_1 a_2, A, A) = (a_1 a_2, d_1 d_2, A)
  \]

Next we consider location \( l_6 \).

• \( \text{vec.map}(l_6)^I = \text{vec.map}(l_5) = (a_1, d_1 d_2, c_1) \)

• \( \text{vec.map}(l_6)^{II} = \text{vec.map}(l_{10}) = (a_1 a_2, A, A) \)
  since \( l_{10} \) is the only location in the 2nd operand of \( \text{par} \)

• \( \text{vec.map}(l_6)^{III} = \mathcal{U}_{l_6,l_{10}} \)
  since \( l_{10} \) is the only location of the 2nd operand of \( \text{par} \). The corresponding component vector is given by
  \[
  \mathcal{U}_{l_6,l_{10}} = \mathcal{U}_0 \cup \mathcal{U}_{l_{10}} = (a_1, d_1 d_2, c_1) \sqcup (a_1 a_2, A, A) = (a_1 a_2, d_1 d_2, c_1)
  \]

The cases \( I, II, III \) for location \( l_7 \) are as follows.

• \( \text{vec.map}(l_7)^I = \text{vec.map}(l_7) = (a_1, d_1, e_2) \)

• \( \text{vec.map}(l_7)^{II} = \text{vec.map}(l_{10}) = (a_1 a_2, A, A) \)
  since \( l_{10} \) is the only location in the 2nd operand of \( \text{par} \)

• \( \text{vec.map}(l_7)^{III} = \mathcal{U}_{l_7,l_{10}} \)
  since \( l_{10} \) is the only location of the 2nd operand of \( \text{par} \). The corresponding component vector is given by
  \[
  \mathcal{U}_{l_7,l_{10}} = \mathcal{U}_0 \cup \mathcal{U}_{l_{10}} = (a_1, d_1, e_2) \sqcup (a_1 a_2, A, A) = (a_1 a_2, d_1, e_2)
  \]

We next consider location \( l_8 \).

• \( \text{vec.map}(l_8)^I = \text{vec.map}(l_8) = (a_1, d_1 d_3, e_2) \)

• \( \text{vec.map}(l_8)^{II} = \text{vec.map}(l_{10}) = (a_1 a_2, A, A) \)
  since \( l_{10} \) is the only location in the 2nd operand of \( \text{par} \)
• vec.map(l₀)_{III} = \mathcal{U}_{l₀,i₁₀} \\
  
since \ l₁₀ is the only location of the 2nd operand of par. The corresponding 
component vector is given by 
\mathcal{U}_{l₀,i₁₀} = \mathcal{U}_{l₀,i₁₀} \cup \mathcal{U}_{l₁₀} = (a₁, d₁d₃, e₂) \cup (a₁a₂, \Lambda, \Lambda) = (a₁a₂, d₁d₃, e₂) \\

We continue with location \ l₀. Recall that this location is the end location of the alt 
fragment with 2 operands and thus has been associated with two component vectors 
\mathcal{U}_{l₀}^{(1)} = (a₁, d₁d₂, e₁) and \mathcal{U}_{l₀}^{(2)} = (a₁, d₁d₃, e₂).

• vec.map(l₀)^I = vec.map(l₀) = \{(a₁, d₁d₂, e₁), (a₁, d₁d₃, e₂)\} \\

• vec.map(l₀)^{III} = vec.map(l₁₀) = (a₁a₂, \Lambda, \Lambda) \\
  
since \ l₁₀ is the only location in the 2nd operand of par

• vec.map(l₀)^{III} = \mathcal{U}_{l₁₀}^{(1)} \cup \mathcal{U}_{l₁₀}^{(2)} \\
  
since \ l₁₀ is the only location of the 2nd operand of par. The corresponding 
component vectors are given by 
\mathcal{U}_{l₀,i₁₀}^{(1)} = \mathcal{U}_{l₀}^{(1)} \cup \mathcal{U}_{l₁₀}^{(1)} = (a₁, d₁d₂, e₁) \cup (a₁a₂, \Lambda, \Lambda) = (a₁a₂, d₁d₂, e₁) \\
and 
\mathcal{U}_{l₀,i₁₀}^{(2)} = \mathcal{U}_{l₀}^{(2)} \cup \mathcal{U}_{l₁₀}^{(2)} = (a₁, d₁d₃, e₂) \cup (a₁a₂, \Lambda, \Lambda) = (a₁a₂, d₁d₃, e₂) \\

Thus, vec.map(l₀)^{III} = \{(a₁a₂, d₁d₂, e₁), (a₁a₂, d₁d₃, e₂)\}

Note that we do not apply the cases I, II, III construction to location \ l₁₀ although it 
is in par because it belongs to the last operand of the fragment, i.e. \ l₁₀ ∈ Loc_{par} but 
\text{scope}(l₁₀) = npa.par(2)\%2.

Now location \ l₁₁ is the end location of par. Thus, the component vectors associated 
with \ l₁₁ are given by Definition 3.2.6 as follows.

vec.map(l₁₁) = vec.map(l₃)^{x} \cup vec.map(l₄)^{x} \cup vec.map(l₅)^{x} \cup vec.map(l₆)^{x} \cup 
vec.map(l₇)^{x} \cup vec.map(l₈)^{x} \cup vec.map(l₉)^{x}, \ x = I, II, III

\mathcal{U}_{l₁₁} = \mathcal{U}_{l₁₁} \cup (a₁a₂, d₁d₃, e₂) \cup (a₁a₂, d₁d₃, e₂) \cup (a₁a₂, d₁d₃, e₂) \cup (a₁a₂, d₁d₃, e₂) \cup (a₁a₂, d₁d₃, e₂) \cup (a₁a₂, d₁d₃, e₂) \cup (a₁a₂, d₁d₃, e₂)
Thus,

\[
\text{vec.map}(l_{11}) = \{(a1, d1, A), (a1a2, A, A), (a1a2, d1, A), (a1, d1d2, A), (a1a2, d1d2, A), \\
(a1, d1d2, e1), (a1a2, d1d2, e1), (a1, d1, e2), (a1a2, d1, e2), \\
(a1, d1d3, e2), (a1a2, d1d3, e2)\}
\]

Having reached the final location \( l_{12} \) of the diagram, we may obtain the component language of \( B \) by applying Definition 3.2.7. The definition says that this is given by the union of the component vectors mapped onto all locations of the diagram that are not within a par or a sim interaction fragment. In this case, these are \( l_0, l_1, l_2 \) and \( l_{11} \). Thus, we have

\[
V_B = \{(A, A, A), (a1, A, A), (a1, d1, A), (a1a2, A, A), (a1a2, d1, A), (a1, d1d2, A), \\
(a1a2, d1d2, A), (a1, d1d2, e1), (a1a2, d1d2, e2), (a1, d1, e2), (a1a2, d1, e2), \\
(a1, d1d3, e2), (a1a2, d1d3, e2)\}
\]

The sequence appearing on a coordinate of a component vector in \( V_B \) describes the ordering of events associated with the interface corresponding to that coordinate. The orderings between component vectors describe the relationships between events associated with different interfaces of the component \( B \).

The order structure of the component language is shown in the Hasse diagram of Figure A.2. It can be seen that there are five separate diamonds (one is relatively prolonged and dashed) that indicate concurrency on the interfaces of component \( B \).

Starting from bottom left and continuing clock-wise, the first diamond indicates concurrency between events \( d1 \) and \( a2 \), the next one between \( d2 \) and \( a2 \), the next between \( e1 \) and \( a2 \), the next between \( d3 \) and \( a2 \), and the last one (which is partly dashed and prolonged) indicates concurrency between \( e2 \) and \( a2 \). The dashed lines forming the last diamond and leading to the top right one, also indicate an ordering between vectors just like the solid ones. We use them to indicate that after the diamond for \( d1, a2 \) there is a choice between the diamond for \( d2, a2 \) (which enables the subsequent diamond for \( e1, a2 \)) and the diamond for \( e1, a2 \) (which enables the subsequent diamond for \( d3, a2 \)).
Figure A.2: The order structure of the component language $V_B$ of component $B$

This choice is specified in the sequence diagram of Figure A.1 using the nested alt interaction fragment.

In further explanation, the three adjacent solid-line diamonds reflect the ordering between the events within the par fragment appearing in the sequence diagram of Figure A.1 when the scenario described in the 1st operand of the nested alt is executed. The bottom left, the dashed and the top right adjacent diamonds reflect the ordering between events when the 2nd operand of the nested alt fragment is taken. It might be instructive to look back to the sequence diagram and examine the possible sequences of executions through the par and nested alt fragments until reaching the end of the diagram. These are reflected in the Hasse diagram, with the additional provision of concurrency with a2 in each case.

It can also be seen from the Hasse diagram of Figure A.2 that $V_B$ is discrete - its order structure exhibits the characteristic structure of a lattice. Any two vectors that are bounded above (i.e. are less or equal than another vector) in $V_B$, have their least upper bound and greatest lower bound in $V_B$. In addition, the empty vector $(\lambda, \lambda, \lambda)$ is in the language. The component language $V_B$ is also locally left-closed. By inspection,
Appendix A. Nested locations in UML2.0 sequence diagrams: an example

It may be seen that what takes us from one vector to its immediate successor is the occurrence of a single event per interface. Hence, the sequences on coordinates of component vectors are built up by adding one event at a time. Thus, we may deduce that component $B$ is well-behaved and on this evidence its scenario-based specification given in the sequence diagram of Figure A.1 does not contain race conditions.
Appendix B

Index of Terms and Symbols

In this appendix we list the most significant terms and symbols used in the thesis along with the page numbers on which they first appear. We start with the terms, in alphabetical order. Then we give the symbols, also in alphabetical order, starting with the English letters used, then the Greek letters and then the mathematical symbols.

Terms

alternative locations p. 107
assembly connector p. 209
associative p. 207
behaviour protocols p. 56
behavioural presentation p. 229
coherent p. 233
column vector p. 85, p. 165
commutative p. 204
compatible automata p. 277
compatible components p. 212
component p. 1
component language p. 87
component lifeline p. 95
Appendix B. Index of Terms and Symbols

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>component signature</td>
<td>p. 80</td>
</tr>
<tr>
<td>component technologies</td>
<td>p. 13</td>
</tr>
<tr>
<td>component vector</td>
<td>p. 82</td>
</tr>
<tr>
<td>composite component</td>
<td>p. 203</td>
</tr>
<tr>
<td>composite signature</td>
<td>p. 200</td>
</tr>
<tr>
<td>composite structure diagram</td>
<td>p. 35</td>
</tr>
<tr>
<td>composition</td>
<td>p. 196</td>
</tr>
<tr>
<td>compound transition</td>
<td>p. 266</td>
</tr>
<tr>
<td>concatenation</td>
<td>p. 155</td>
</tr>
<tr>
<td>concurrency</td>
<td>p. 4</td>
</tr>
<tr>
<td>concurrent events</td>
<td>p. 170</td>
</tr>
<tr>
<td>consecutive events</td>
<td>p. 172</td>
</tr>
<tr>
<td>consistent component vectors</td>
<td>p. 202</td>
</tr>
<tr>
<td>consistent signatures</td>
<td>p. 198</td>
</tr>
<tr>
<td>consistently complete</td>
<td>p. 161</td>
</tr>
<tr>
<td>coordinate-wise concatenation</td>
<td>p. 85, p. 155</td>
</tr>
<tr>
<td>decomposition</td>
<td>p. 199</td>
</tr>
<tr>
<td>delegation connector</td>
<td>p. 210</td>
</tr>
<tr>
<td>dependencies</td>
<td>p. 3</td>
</tr>
<tr>
<td>diamond</td>
<td>p. 173</td>
</tr>
<tr>
<td>discrete component language</td>
<td>p. 161</td>
</tr>
<tr>
<td>discrete behavioural presentation</td>
<td>p. 238</td>
</tr>
<tr>
<td>dynamic characteristics</td>
<td>p. 82</td>
</tr>
<tr>
<td>Einstein’s thought experiment</td>
<td>p. 170</td>
</tr>
<tr>
<td>empty vector</td>
<td>p. 84</td>
</tr>
<tr>
<td>end location</td>
<td>p. 97</td>
</tr>
<tr>
<td>equivalence class</td>
<td>p. 235</td>
</tr>
<tr>
<td>event</td>
<td>p. 76</td>
</tr>
<tr>
<td>events in conflict</td>
<td>p. 176</td>
</tr>
<tr>
<td>execution vector</td>
<td>p. 258</td>
</tr>
<tr>
<td>Term</td>
<td>Page(s)</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>Finite State Machine (FSM)</td>
<td>p. 63</td>
</tr>
<tr>
<td>first location of each operand</td>
<td>p. 108</td>
</tr>
<tr>
<td>fork</td>
<td>p. 266</td>
</tr>
<tr>
<td>greatest lower bound</td>
<td>p. 158</td>
</tr>
<tr>
<td>Hasse diagram</td>
<td>p. 168</td>
</tr>
<tr>
<td>high-level MSC</td>
<td>p. 45</td>
</tr>
<tr>
<td>horizontal communication interfaces</td>
<td>p. 17</td>
</tr>
<tr>
<td>independent vectors</td>
<td>p. 172</td>
</tr>
<tr>
<td>inside-out approach</td>
<td>p. 125</td>
</tr>
<tr>
<td>intended behaviour</td>
<td>p. 87</td>
</tr>
<tr>
<td>interaction</td>
<td>p. 37</td>
</tr>
<tr>
<td>interaction fragment</td>
<td>p. 89</td>
</tr>
<tr>
<td>interaction operator</td>
<td>p. 89</td>
</tr>
<tr>
<td>Interaction Overview Diagram (IOD)</td>
<td>p. 45</td>
</tr>
<tr>
<td>interface</td>
<td>p. 1</td>
</tr>
<tr>
<td>interface automata</td>
<td>p. 69</td>
</tr>
<tr>
<td>irreflexivity</td>
<td>p. 231</td>
</tr>
<tr>
<td>join</td>
<td>p. 266</td>
</tr>
<tr>
<td>Koala</td>
<td>p. 15</td>
</tr>
<tr>
<td>last location of each operand</td>
<td>p. 107</td>
</tr>
<tr>
<td>least upper bound</td>
<td>p. 158</td>
</tr>
<tr>
<td>left-closed</td>
<td>p. 237</td>
</tr>
<tr>
<td>lifeline</td>
<td>p. 37</td>
</tr>
<tr>
<td>Live Sequence Chart (LSC)</td>
<td>p. 39</td>
</tr>
<tr>
<td>local concurrency</td>
<td>p. 262</td>
</tr>
<tr>
<td>locally left-closed component language</td>
<td>p. 163</td>
</tr>
<tr>
<td>location</td>
<td>p. 89</td>
</tr>
<tr>
<td>lower diamond property</td>
<td>p. 270</td>
</tr>
<tr>
<td>lozenge rule</td>
<td>p. 263</td>
</tr>
<tr>
<td>maximal component vectors</td>
<td>p. 133, p. 156</td>
</tr>
</tbody>
</table>
### Appendix B. Index of Terms and Symbols

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mazurkiewicz traces</td>
<td>171</td>
</tr>
<tr>
<td>Message Sequence Chart (MSC)</td>
<td>38</td>
</tr>
<tr>
<td>mutually exclusive events</td>
<td>231</td>
</tr>
<tr>
<td>nested locations</td>
<td>125</td>
</tr>
<tr>
<td>normal component language</td>
<td>163</td>
</tr>
<tr>
<td>n-tuple</td>
<td>82</td>
</tr>
<tr>
<td>operand</td>
<td>89</td>
</tr>
<tr>
<td>orthogonal region</td>
<td>266</td>
</tr>
<tr>
<td>pair-wise compatible</td>
<td>233</td>
</tr>
<tr>
<td>parallel construct</td>
<td>42</td>
</tr>
<tr>
<td>parallel locations</td>
<td>113</td>
</tr>
<tr>
<td>pathological behaviour</td>
<td>3</td>
</tr>
<tr>
<td>pattern of behaviour</td>
<td>134</td>
</tr>
<tr>
<td>permute</td>
<td>172</td>
</tr>
<tr>
<td>points</td>
<td>229</td>
</tr>
<tr>
<td>prefix ordering</td>
<td>155</td>
</tr>
<tr>
<td>pre-order</td>
<td>232</td>
</tr>
<tr>
<td>preservation under composition</td>
<td>211</td>
</tr>
<tr>
<td>primal vector</td>
<td>242</td>
</tr>
<tr>
<td>prime</td>
<td>240</td>
</tr>
<tr>
<td>prime algebraic</td>
<td>240</td>
</tr>
<tr>
<td>provided interface</td>
<td>1, 79</td>
</tr>
<tr>
<td>race conditions</td>
<td>38, 186</td>
</tr>
<tr>
<td>reflexivity</td>
<td>232</td>
</tr>
<tr>
<td>replete</td>
<td>235</td>
</tr>
<tr>
<td>required interface</td>
<td>1, 79</td>
</tr>
<tr>
<td>right-cancellation</td>
<td>159</td>
</tr>
<tr>
<td>scenario</td>
<td>37</td>
</tr>
<tr>
<td>sequential locations</td>
<td>101</td>
</tr>
<tr>
<td>simultaneity class</td>
<td>165</td>
</tr>
<tr>
<td>Term</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------</td>
</tr>
<tr>
<td>simultaneity events</td>
<td>85</td>
</tr>
<tr>
<td>simultaneous locations</td>
<td>121</td>
</tr>
<tr>
<td>snapshot</td>
<td>100</td>
</tr>
<tr>
<td>start location</td>
<td>97</td>
</tr>
<tr>
<td>static specification</td>
<td>80</td>
</tr>
<tr>
<td>step</td>
<td>234</td>
</tr>
<tr>
<td>streams</td>
<td>51</td>
</tr>
<tr>
<td>symmetric difference</td>
<td>202</td>
</tr>
<tr>
<td>symmetry</td>
<td>231</td>
</tr>
<tr>
<td>temporal relations</td>
<td>230</td>
</tr>
<tr>
<td>transition relation</td>
<td>257</td>
</tr>
<tr>
<td>transitivity</td>
<td>232</td>
</tr>
<tr>
<td>UML component</td>
<td>28, 32</td>
</tr>
<tr>
<td>UML metamodel</td>
<td>25</td>
</tr>
<tr>
<td>Unified Modeling Language (UML)</td>
<td>20</td>
</tr>
<tr>
<td>vertical control interfaces</td>
<td>17</td>
</tr>
<tr>
<td>well-behaved component</td>
<td>163</td>
</tr>
<tr>
<td>Zeno paradoxes</td>
<td>234</td>
</tr>
<tr>
<td>$\Sigma$-automaton</td>
<td>263</td>
</tr>
<tr>
<td>$\Sigma$-machine</td>
<td>257</td>
</tr>
</tbody>
</table>

**Symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt</td>
<td>90</td>
</tr>
<tr>
<td>$alt(n)$</td>
<td>97</td>
</tr>
<tr>
<td>$\overline{alt(n)}$</td>
<td>98</td>
</tr>
<tr>
<td>$alt(n)|k$</td>
<td>97</td>
</tr>
<tr>
<td>$B$</td>
<td>230</td>
</tr>
<tr>
<td>$base(y)$</td>
<td>242</td>
</tr>
<tr>
<td>$c$</td>
<td>87</td>
</tr>
<tr>
<td>$Cline$</td>
<td>95</td>
</tr>
</tbody>
</table>
## Appendix B. Index of Terms and Symbols

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$co$</td>
<td>230</td>
</tr>
<tr>
<td>$cov(y)$</td>
<td>241</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>165</td>
</tr>
<tr>
<td>$\varepsilon(i)$</td>
<td>165</td>
</tr>
<tr>
<td>$E$</td>
<td>230</td>
</tr>
<tr>
<td>$E_{\nu}$</td>
<td>165</td>
</tr>
<tr>
<td>$E_{\nu}^\perp$</td>
<td>165</td>
</tr>
<tr>
<td>$E(M)$</td>
<td>276</td>
</tr>
<tr>
<td>$i$</td>
<td>80</td>
</tr>
<tr>
<td>$I$</td>
<td>80</td>
</tr>
<tr>
<td>$ind$</td>
<td>172</td>
</tr>
<tr>
<td>$I_q^M$</td>
<td>261</td>
</tr>
<tr>
<td>$I_E$</td>
<td>80</td>
</tr>
<tr>
<td>$l$</td>
<td>96</td>
</tr>
<tr>
<td>$l_0$</td>
<td>95</td>
</tr>
<tr>
<td>$Loc$</td>
<td>95</td>
</tr>
<tr>
<td>$Loc_{alt}$</td>
<td>100</td>
</tr>
<tr>
<td>$Loc_c$</td>
<td>100</td>
</tr>
<tr>
<td>$Loc_{end_alt}$</td>
<td>99</td>
</tr>
<tr>
<td>$Loc_{end_par}$</td>
<td>99</td>
</tr>
<tr>
<td>$Loc_{end_sim}$</td>
<td>99</td>
</tr>
<tr>
<td>$Loc_{par}$</td>
<td>100</td>
</tr>
<tr>
<td>$Loc_{seq}$</td>
<td>100</td>
</tr>
<tr>
<td>$Loc_{sim}$</td>
<td>100</td>
</tr>
<tr>
<td>$Loc_{start_alt}$</td>
<td>99</td>
</tr>
<tr>
<td>$Loc_{start_par}$</td>
<td>99</td>
</tr>
<tr>
<td>$Loc_{start_sim}$</td>
<td>99</td>
</tr>
<tr>
<td>$M$</td>
<td>257</td>
</tr>
<tr>
<td>$M_c$</td>
<td>259</td>
</tr>
<tr>
<td>$M_c^*$</td>
<td>259</td>
</tr>
</tbody>
</table>
\[(M_c, \Delta_r)\] p. 259
max\((u(i), v(i))\) p. 158
min\((u(i), v(i))\) p. 158
\(N_0\) p. 96
\(N^+\) p. 97
\(O\) p. 231
\(Op\) p. 80
\(Op_{\text{II}}\) p. 80
\(Op_{\text{III}}\) p. 82
\(P\) p. 80
\(Path\) p. 95
\(par\) p. 91
\(par(n)\) p. 97
\(\overline{par(n)}\) p. 98
\(par(n)\#k\) p. 97
\(prmi(V)\) p. 242
\(q\) p. 257
\(q\) p. 274
\(Q\) p. 257
\(R\) p. 80
\(RE\) p. 95
\(scope\) p. 97
\(sd\) p. 97
\(SE\) p. 95
\(sim\) p. 93
\(\overline{sim}\) p. 98
\(\overline{sim\#1}\) p. 98
\(time\) p. 96
\(u_1 | f_{x_1} \cap f_{x_2}\) p. 202
\(v\) p. 82
### Appendix B. Index of Terms and Symbols

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ω(i)</td>
<td>82</td>
</tr>
<tr>
<td>ω[k]</td>
<td>274</td>
</tr>
<tr>
<td>ω_{i}^{(k)}</td>
<td>104</td>
</tr>
<tr>
<td>vec.map(l)</td>
<td>103</td>
</tr>
<tr>
<td>vec.map(l)_{I}</td>
<td>116</td>
</tr>
<tr>
<td>vec.map(l)_{II}</td>
<td>116</td>
</tr>
<tr>
<td>vec.map(l)_{III}</td>
<td>116</td>
</tr>
<tr>
<td>vec.map(l_{r})^{z}</td>
<td>117</td>
</tr>
<tr>
<td>V</td>
<td>87</td>
</tr>
<tr>
<td>V(M,q)</td>
<td>269</td>
</tr>
<tr>
<td>V_{Σ}</td>
<td>82</td>
</tr>
<tr>
<td>XΔY</td>
<td>202</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Symbol</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>$\equiv_1^{(1)}$</td>
<td>172</td>
</tr>
<tr>
<td>$\equiv_2$</td>
<td>172</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>198</td>
</tr>
<tr>
<td>$\Theta$</td>
<td>200</td>
</tr>
<tr>
<td>$\parallel$</td>
<td>230</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>230</td>
</tr>
<tr>
<td>$\equiv$</td>
<td>230</td>
</tr>
<tr>
<td>$\vdash$</td>
<td>257</td>
</tr>
<tr>
<td>$\Rightarrow$</td>
<td>274</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>276</td>
</tr>
</tbody>
</table>
Appendix B. Index of Terms and Symbols
Bibliography


<table>
<thead>
<tr>
<th>References</th>
<th>Details</th>
</tr>
</thead>
</table>


