Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.
Foundations for Semantically Enhanced Component Trading
A Component Type Model

By
Sotirios Terzis

Submitted for the degree of Doctor of Philosophy to the Department of Computer Science, University of Dublin, Trinity College

2004
I declare that
(a) this thesis has not been submitted as an exercise for a degree at this or any other University, and
(b) the work described in this thesis is exclusively my own.

I agree that the Library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgment.

Sotirios Terzis.

Dublin, September 2004.
Acknowledgements

This thesis would never have been completed without the continuous support and encouragement of my supervisor Paddy Nixon. I sincerely thank him for this and for not loosing faith in me during this particularly long process.

Moreover, I would like to express my gratitude to my parents, Giannis and Maria for their numerous sacrifices in order to provide me with the best possible education. I would also like to thank them for their continuous moral support over the years and in particular their encouragement over the last few weeks.

Furthermore, I would like to express my gratitude to Ann for her love, support and understanding during the writing up stage of this thesis.

In addition to the above, I would also like to thank Christos Nikolaou for encouraging me to pursue a Ph.D., my friend Manolis Marazakis for continuously pressing me to complete this thesis, and Ken Moody for his generous offer for the writing up process.

Finally, I would also like to thank the fellow students and colleagues of the Distributed Systems Group of the Computer Science Department, Trinity College Dublin and the Computer and Information Sciences Department, University of Strathclyde for their help and support over these years.
Summary

This thesis introduces the notion of Semantically Enhanced Component Trading (SECT) in order to bring the notion of service discovery, widely used in large-scale networked and distributed systems, into the domain of component-based software development, and while doing so, to also improve it by providing it at a higher quality than currently available.

The way in which SECT enhances the notion of service trading is by introducing as a foundation to the basic trading process a component type model based on semantic notions of component types and component conformance. This component type model is the primary focus of this thesis. In contrast to type models for service trading, the component type model for SECT requires type definitions that include explicit descriptions of type behaviour, i.e. semantics. These descriptions of type semantics are in terms of pre-/post-conditions, invariants, history constraints and model programs, appropriately extended to cover the specification of component types providing and requiring a number of services. At the same time, the type model for SECT utilises ideas from signature and specification based matching, as well as behavioural subtyping, to define a rich set of type compatibility/conformance relationships able to enhance the trading process. Moreover, recognising the difficulties of establishing semantic type relationships, the use of contextual composition in current component platform, and the standardisation efforts currently underway in certain application domains, the component type model for SECT also introduces a preliminary notion of type definition domains. These domains may be used to standardise particular type definitions, models for the description of extra-functional properties, and certain terminology used in particular application domains. All these elements of type definition domains may also be utilised in order to streamline the operation of type managers supporting the component type model for SECT.

A number of example component type definitions are used to demonstrate the preciseness and expressiveness of the SECT component type model. At the same time, a SECT-enabled trader architecture, which takes advantage of the particular features of the
component type model, is used as a vehicle to explore the issues pertaining to the implementation of a SECT-enabled trader.
Table of contents

Foundations for Semantically Enhanced Component Trading ................................................. 1
Acknowledgements ...................................................................................................................... 5
Summary ........................................................................................................................................ 6
Table of contents ............................................................................................................................. 9
Table of figures .................................................................................................................................. 13
Chapter 1. Introduction .................................................................................................................. 15
  1.1. Motivation .................................................................................................................................. 15
    1.1.1. The virtual enterprise model ............................................................................................ 16
    1.1.2. Facing up to the challenges of the virtual enterprise model ........................................... 19
        A note on the feasibility of component-orientation ................................................................. 20
        A component based enterprise information system architecture ........................................... 22
        Concluding remarks ................................................................................................................ 25
  1.2. Problem statement .................................................................................................................. 26
  1.3. Approach overview ............................................................................................................... 27
Chapter 2. Component software .................................................................................................. 31
  2.1. A brief overview of component software research ................................................................. 31
  2.2. What are software components? ............................................................................................. 36
      2.2.1. Software components: binary or not ............................................................................. 37
      2.2.2. Software components: units of independent deployment and third party composition .................................................................................................................. 39
      2.2.3. Software components: stateless or stateful ................................................................. 41
      2.2.4. Software components: component models, component architecture and standards ................................................................. 44
      2.2.5. COTS components ...................................................................................................... 46
  2.3. Component platforms: current state ..................................................................................... 48
      2.3.1. The Java perspective: JavaBeans and Enterprise JavaBeans ........................................ 50
      The Java language .................................................................................................................. 51
      The Java component models ................................................................................................. 53
      2.3.2. The OMG perspective: CORBA and CCM .................................................................. 58
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.7</td>
<td>Summary</td>
<td>127</td>
</tr>
<tr>
<td>4.3</td>
<td>Type naming</td>
<td>128</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Definition of our type naming scheme</td>
<td>129</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Trader federation and type naming</td>
<td>132</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Summary</td>
<td>132</td>
</tr>
<tr>
<td>4.4</td>
<td>Type description</td>
<td>133</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Describing models of type behaviour in UML/OCL</td>
<td>133</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Component type description</td>
<td>136</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Service type description</td>
<td>139</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Action type description</td>
<td>141</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Property type description</td>
<td>142</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Component and service behavioural interface type description</td>
<td>142</td>
</tr>
<tr>
<td>4.4.7</td>
<td>Component, service and action interface type description</td>
<td>143</td>
</tr>
<tr>
<td>4.4.8</td>
<td>Basic types, constructed types and data types description</td>
<td>144</td>
</tr>
<tr>
<td>4.4.9</td>
<td>Summary</td>
<td>144</td>
</tr>
<tr>
<td>4.5</td>
<td>Type relationships</td>
<td>145</td>
</tr>
<tr>
<td>4.5.1</td>
<td>A framework for type relation definition based on object-oriented typing</td>
<td>146</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Refining the framework within the solution context</td>
<td>149</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Name-based compatibility/conformance</td>
<td>151</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Structure-based compatibility/conformance</td>
<td>153</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Syntactic compatibility/conformance</td>
<td>158</td>
</tr>
<tr>
<td>4.5.6</td>
<td>Semantics-based compatibility/conformance</td>
<td>158</td>
</tr>
<tr>
<td>4.5.7</td>
<td>Type relationship specification</td>
<td>166</td>
</tr>
<tr>
<td>4.5.8</td>
<td>Summary</td>
<td>167</td>
</tr>
<tr>
<td>4.6</td>
<td>Polymorphism and inheritance</td>
<td>168</td>
</tr>
<tr>
<td>4.7</td>
<td>Type definition domains</td>
<td>170</td>
</tr>
<tr>
<td>4.8</td>
<td>Concluding remarks</td>
<td>172</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>SECT component type model evaluation</td>
<td>175</td>
</tr>
<tr>
<td>5.1</td>
<td>Example SECT component type model type definitions</td>
<td>175</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Looking into the relationships between example types</td>
<td>181</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Looking into more elaborate behavioural specifications</td>
<td>190</td>
</tr>
</tbody>
</table>
# Table of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The Virtual Enterprise a worldwide collection of independent performers</td>
<td>16</td>
</tr>
<tr>
<td>1.2</td>
<td>Current modes of interaction for enterprise information systems</td>
<td>18</td>
</tr>
<tr>
<td>1.3</td>
<td>Enterprise information system interaction for dynamic collaboration</td>
<td>19</td>
</tr>
<tr>
<td>1.4</td>
<td>Enterprise System Architecture: Management View</td>
<td>22</td>
</tr>
<tr>
<td>1.5</td>
<td>Enterprise System Architecture: Systems Level View (Components)</td>
<td>23</td>
</tr>
<tr>
<td>1.6</td>
<td>Enterprise System Architecture: System Level View (Enterprise Services)</td>
<td>24</td>
</tr>
<tr>
<td>2.1</td>
<td>J2EE architectural overview</td>
<td>50</td>
</tr>
<tr>
<td>2.2</td>
<td>Client/Server interaction in CORBA</td>
<td>59</td>
</tr>
<tr>
<td>2.3</td>
<td>The Structure of COM components</td>
<td>64</td>
</tr>
<tr>
<td>2.4</td>
<td>COM Component with multiple interfaces and objects</td>
<td>65</td>
</tr>
<tr>
<td>3.1</td>
<td>Trading Architecture</td>
<td>87</td>
</tr>
<tr>
<td>4.1</td>
<td>A Meta-Object Facility meta-model of the SECT type model</td>
<td>111</td>
</tr>
<tr>
<td>4.2</td>
<td>Compatibility Classes</td>
<td>151</td>
</tr>
<tr>
<td>4.3</td>
<td>Syntactic conformance</td>
<td>151</td>
</tr>
<tr>
<td>4.4</td>
<td>Relating the function specification matches</td>
<td>161</td>
</tr>
<tr>
<td>4.5</td>
<td>Selected Function Specification Matches</td>
<td>163</td>
</tr>
<tr>
<td>5.1</td>
<td>BasicAccount service behaviour model (BAcc_BM)</td>
<td>177</td>
</tr>
<tr>
<td>5.2</td>
<td>IPBasicAccount service behaviour model (IBAcc_BM)</td>
<td>182</td>
</tr>
<tr>
<td>5.3</td>
<td>FRBasicAccount behavioural model (FBAcc_BM)</td>
<td>185</td>
</tr>
<tr>
<td>5.4</td>
<td>OBasicAccount behavioural model (OAcc_BM)</td>
<td>186</td>
</tr>
<tr>
<td>5.5</td>
<td>BasicCredit behavioural model (BCredit_BM)</td>
<td>187</td>
</tr>
<tr>
<td>5.6</td>
<td>CardAwareAccount behavioural model (CAcc_BM)</td>
<td>190</td>
</tr>
<tr>
<td>5.7</td>
<td>SwitchCard component behaviour model (SCard_BM)</td>
<td>193</td>
</tr>
<tr>
<td>5.8</td>
<td>High-level SECT-enabled trader architecture</td>
<td>197</td>
</tr>
<tr>
<td>5.9</td>
<td>Request processor architecture</td>
<td>202</td>
</tr>
<tr>
<td>5.10</td>
<td>Offer manager architecture</td>
<td>205</td>
</tr>
<tr>
<td>5.11</td>
<td>Type manager architecture</td>
<td>205</td>
</tr>
<tr>
<td>5.12</td>
<td>Composition facility architecture</td>
<td>212</td>
</tr>
<tr>
<td>5.13</td>
<td>Enterprise information system development</td>
<td>214</td>
</tr>
</tbody>
</table>
Chapter 1. Introduction

1.1. Motivation

Over the years enterprises have recognised the potential and the opportunities information and communication technologies offer, both in terms of expanding their business and in terms of streamlining their internal operations. As a result, these technologies have become an integral part of most enterprises, usually in the form of an enterprise information system. It is quite commonplace nowadays for these information systems to occupy a central place within the enterprise playing a crucial role in its success. Considering the serious impact these information systems have in the operation of the enterprise, it comes as no surprise that the ways in which enterprises are organised, and the advances in information and communication technologies have been moving in close relationship to each other.

Examining the relationship between enterprise organisational models and advances in information and communication technologies closer, we observe the following cycle. The changing needs of enterprises lead to the development of new organisational models that are better suited in satisfying these needs. These emerging organisational models present enterprise information systems with a series of challenges that require certain advances in the underlying information and communication technologies used. The incorporation of these developments into the enterprise information systems enables new modes of enterprise operation that open new business opportunities. These opportunities bring further changes to the needs of the enterprise that in turn result to the development of newer organisational models.

Over the past few years, we have been witnessing the emergence of the Virtual Enterprise (VE) model for business organisations initiate another round of the cycle. This new round is also being driven by a combination of business needs and technological advances. We are currently at the point of the cycle where the emphasis shifts from the development of the underlying information and communication technologies to the development of enterprise information systems that utilise these technologies. As a result, the focus is currently on devising an approach to enterprise information system
development that is able to both incorporate the recent technological advances and satisfy the requirements of the VE model. A component oriented software development approach seems particularly promising for this purpose. However, in order to understand and appreciate its potential, it is important to take a closer look at the VE model.

1.1.1. The virtual enterprise model

Over the past few years, business practices have undergone a dramatic change. With the deregulation of markets and the opening of borders for trade and services, there is a growing need for businesses to form associations and partnerships to best exploit market opportunities and to timely deliver products and services. Operating in a business environment dominated by accelerating product cycles and improved product targeting implies that both fundamental (long-term) and market-driven (short-term) collaborations are critical to a business' continued competitiveness [58]. Moreover, downsizing and narrow niche markets suggest that collaborations have great potential for reducing costs within the global marketplace. In such a business environment where market opportunities emerge rapidly and profit margins are tight, the formation of business collaborations in response to the changing market condition is the key to success. The VE model for business organisations has been devised to address precisely this need for agility within the new business environment.

![Building a Worldwide Team](image)

Figure 1.1. The Virtual Enterprise a worldwide collection of independent performers.
A virtual enterprise is defined as a collaborative association between administratively and possibly geographically distributed business units. It consists of a set of legally independent performers of varying types who voluntarily cooperate to seize particular market opportunities. They are represented by at least one partner to the external world and they agree to produce a common output, a product or a service, based on a common understanding of their business rules and business processes. Within a virtual enterprise environment, performers share resources, core competencies, skills, and know how in order to become quicker, more flexible and more global (see Figure 1.1). As a result, the operations of a virtual enterprise are defined as a set of business processes shared according to well-defined contracts and agreements.

Appel in [59] has investigated a number of virtual enterprises in an attempt to identify their defining characteristics. His investigation uncovered two key structural characteristics of them: (a) interdependence between constituent operations and (b) distribution of responsibility between constituent operations. He also found out that although they can take a number of different forms, from alliances of organisations or individuals, to established decentralised companies, to central companies seeking to adapt, to even single organisations, they always have at least one of the following characteristics: (i) geographic separation, (ii) functional specialisation with separate reporting hierarchies, (iii) transitory membership driven by evolving needs over time, and (iv) separation of production across different time dimensions. Moreover, he points out that contrary to common belief a virtual organisation does not have to be any more temporary than other enterprises. Actually, what makes virtual enterprises different is their strong re-configurability, continuous reconstruction and element substitutability. Furthermore, he also shows that although the literature leaves the impression that virtual enterprises are only businesses, profit-driven types of organisations, impression that is further amplified by the use of the words, corporation or enterprise, and not organisation, this is not really the case. Virtual organisations also operate with a great deal of success in not-for-profit areas too.

In all cases, the agile and reconfigurable collaboration of high efficiency and adaptability that is the virtual enterprise or organisation can only be achieved with the use and application of information and communication technology (ICT). Hence, the use of
ICT is a constitutive feature of the virtual organisation that allows it to be differentiated from other types of networked organisations. The virtual organisation is a network organisation but, in addition to implementing various forms of co-operation, it makes a heavy and critical use of information technology. Therefore, ICT emerges as the primary integrator of the virtual corporation. Only recent developments in network computing and the Internet have made a truly global and efficient virtual enterprise or organisation a viable idea. However, despite their attractiveness and feasibility there are still a number of challenges to overcome, including the extra complexities of cross-border cooperation and contracts, the exposure of key business processes to a hostile Internet, and the need for rapid association of businesses to adapt to, and exploit, market opportunities. As a result, currently virtual enterprises and organisations, where established, are difficult and time consuming to construct and evolve. Making these collaborations more flexible and dynamic is currently the main challenge.

![Current modes of interaction for enterprise information systems.](image)

The current lack of dynamism in enterprise collaborations can be attributed to the inflexibility of their enterprise information systems, and the limitations of the existing software system integration infrastructures [229]. These infrastructures tend to view the enterprise information systems of the collaborators as rigid autonomous entities. Consequently, during the development of the collaborative enterprise system they place particular emphasis in the protection of the autonomy of its constituent systems. This
results in fairly restricted modes of interaction between these parts (see Figure 1.2), which can only provide limited support for enterprise collaboration. Therefore, the currently available software infrastructures are inappropriate for, or are unable to support, the flexibility and dynamism required by the Virtual Enterprise model [61].

1.1.2. Facing up to the challenges of the virtual enterprise model

![Figure 1.3. Enterprise information system interaction for dynamic collaboration.](image)

The ability of enterprises to organise their operations according to the Virtual Enterprise Model is predicated on the development of software infrastructures able to support dynamic and flexible collaborations. The development of such infrastructures relies more on taking a new perspective on building collaborative enterprise information systems, than significant breakthroughs in particular information and communication technologies [61]. From an organisational point of view, this new perspective should enable closer collaboration between enterprises by embedding parts of one organisation into another and blurring the boundaries between their information systems (see Figure 1.3). From a technical point of view, this new perspective on one hand should recognise and reuse useful existing functionality, while on the other hand it should emphasise and support functionality adaptation and reconfiguration.

The decomposition of enterprise information systems into a set of components, including current popular tools and applications will provide the necessary framework within which this new perspective can be developed; a framework supporting the
dynamic composition and configuration of these components and their deployment in a
controlled environment, as well as their on-the-fly re-composition and reconfiguration
[61]. Moreover, such a framework should be based on open and widely accepted
protocols, in order to overcome the limitations of current proprietary solutions. As the
number of these services will be significant, the framework should also be scalable. In
conclusion, the realisation of the Virtual Enterprise model is predicated on adopting a
component-oriented approach for the development of enterprise information system.

A note on the feasibility of component-orientation

So far, we have advocated that a component-oriented approach can address the
challenges that the Virtual Enterprise model poses to enterprise information system
development. However, we have not provided any arguments supporting the feasibility of
such an approach. Providing such arguments is particularly important if we consider that
a component-oriented development approach has been the holy grail of software
engineering since its inception in the 1960s [45]. It is our belief that although the
realisation of the component software vision is not immediately attainable, due to a
number of technological phenomena over the recent past we are nowadays closer than
ever before. These technological phenomena are:

- The widespread deployment and popularity of the Internet and the World
  Wide Web.
- The development and extensive use of machine-independent programming
  languages.
- The advances in object technology, particularly over the past decade.
- The introduction of compositionality elements into popular software products
  and the relative commercial success of certain types of components.

More specifically, the success of the Internet and the Web has not only provided
us with a global shared data space and communication medium, a prerequisite for the
realisation of the Virtual Enterprise vision, but have also focused attention to the issues of
heterogeneity and distribution. In this respect, efforts have been concentrated in
separating the operational and the deployment aspects of software systems and in how to
enable developers to abstract away from the peculiarities of the underlying computing
and networking platform, bringing the issue of interoperability to the forefront. This has
been the main driving force behind the development of machine-independent programming languages as well as the move toward distributed object platforms.

Machine-independent languages rely on the availability of a runtime environment that provides a common virtual platform enforcing the principle of write-once run everywhere software. This principle plays a major role in paving the way towards component software as it enables the same piece of software to be deployed on a variety of hardware platforms. The prime example of a machine-independent language is Java with its virtual machine [63, 64]. The success of Java is not only attributed to its machine-independence, but also to its links to the Web with the provision of applets and servlets and the provision of a number of supporting services like the Java Messaging Service, the Java Transaction service, etc. Another example is the C# language [65] with the common language runtime, a core element of the .NET platform [66]. Both Java and C# are at the vanguard of object-oriented languages and provide evidence for the widespread acceptance of object technology.

The encapsulation of implementation supported by object-oriented languages has enabled interface-based development. Placing the emphasis on the interface instead of the implementation is fundamental for the facilitation of reuse, and a necessary stage on the way towards component software. Interface-based development has been pushed even further by the development of distributed object platforms like the Common Object Request Broker Architecture (CORBA) [24] and the Distributed Component Object Model (DCOM) [67]. The main aim of these platforms is to allow distributed objects to invoke methods on each other and exchange information in a transparent way, overcoming the intricacies of the various machine, system and even language dependencies [69]. Moreover, these platforms have also been instrumental in identifying the requirements for and addressing the issues of supporting service (see for example CORBA object services [68]) currently available in the Java and .NET platforms. These services allow the location of available objects, simplify the management of objects and ease the development of distributed object systems.

The capabilities of distributed object frameworks enabled the introduction of the first commercial components demonstrating the viability of the approach in business terms. The first example of commercially viable software components was Visual Basic
controls (VBX) [70], which later evolved into ActiveX controls [71] supported by COM [72]. These tended to be restricted to graphical user interface widgets that could be incorporated in any application. Following their success the next example was application plug-ins, currently widely deployed in web browsers and office software suites. These plug-ins enable these applications to handle different types of data content within documents (e.g. Excel spreadsheets in Word documents or PDF documents in web pages). Having made the business case for software component these examples were the main driving force behind developments like Enterprise JavaBeans [73] in the Java platform and CORBA Components in the CORBA platform [18]. These developments have been the result of significant research in the area of component software. The results of this research are a good reason to believe that component software is no longer an elusive vision and to be optimistic that it will soon become a reality. This view will be further reinforced in section 2.1, where we examine the current state of this area.

A component based enterprise information system architecture

![Diagram of Enterprise System Architecture: Management View](image)

Figure 1.4. Enterprise System Architecture: Management View.

For any enterprise of key importance to the successful implementation of its information system is the definition of an appropriate software architecture. In the case of a virtual enterprise such architecture emphasises the ability to integrate, share and
manage business processes located inside different business domain boundaries. While at the same time, it facilitates more flexible and dynamic interactions between collaborators. This architecture can be viewed from two perspectives: the management perspective and the systems level perspective [62]. From a management perspective (see Figure 1.4), the architecture enables a coordinator (top left) to manage the set of projects between collaborating partners (bottom right) that make up the virtual enterprise. Each project is defined as a set of business processes (in the centre), each managing a particular aspect of the project, defined as an interaction between a particular grouping of partners. Following the component-oriented development approach, these business processes are in fact configurations of the components that the partners contribute. Note that the components provided by different partners are depicted in different colour. In this management view, the dynamism of collaborations results from the ability of the coordinator to add and remove collaborators, and reconfigure their interactions through the business processes.

![Figure 1.5. Enterprise System Architecture: Systems Level View (Components).](image)

The systems level view handles the problems of technology integration and realises the management view. Following the component-oriented development approach, at this level the skills and competencies of all collaborators are viewed as sets of services embodied in software components. Moreover, at this level the component configurations of the business processes for the various enterprise projects are realised as distributed
workflows. More specifically, the union of the sets of services/components offered by the various partners creates an enterprise component/service space (see Figure 1.5). This space is enriched by the ability to combine available services and components into new composite ones. It is from this space that the workflows, realising the business processes, will select their constituent components. At the same time, the workflows support the participation of the collaborators in the enterprise project, by presenting them with an interaction interface to its constituent business processes. This interface takes the form of a number of services the partners can use to manage the project.

Figure 1.6. Enterprise System Architecture: System Level View (Enterprise Services).

The formation and management of enterprise projects and their constituent workflows requires the support of a number of enterprise services. These services provide access to the enterprise component/service space in a controlled manner and support the construction of the collaborator interaction interface (see Figure 1.6). The functionality they offer is equivalent to the virtual enterprise enabling services proposed by Ouzounis et al. in [60], namely access control, virtual enterprise contract manager, business processes, business process directory, business process broker/trader and event manager.
However, acknowledging the particular characteristics of the component-oriented development approach this functionality is packaged into:

- **Component location and selection service.** This service is responsible for maintaining information about the available services and components in the virtual enterprise component/service space as well as their relationships. Moreover, it uses this information to assist the formation of the business processes by selecting the appropriate components/services and providing information regarding their location to the distributed workflow engine.

- **Grouping facility.** This service builds the collaborator interaction interface as an entry point to the workflows of the project in the form of a set of services.

- **Distributed workflow engine.** This service is responsible for the activation, monitoring and overall management of the various enterprise workflows during their execution.

- **Contract manager.** Manages, monitors and enforces all virtual enterprise partner contracts. A partner contract specifies the terms of usage for the services the partner provides to other partners.

- **Access control service.** Ensures the authorised access to services. It is a general purpose authentication and authorisation service that enforces security policies during component/service interactions.

In this systems level view, the flexibility of the collaborations results from both the ability to define alternative workflows for the realisation of a business process, and the ability of each workflow to use alternative components in its configuration. Although, all enterprise services play a role in supporting collaboration flexibility, the capability of the component location and selection service is instrumental.

**Concluding remarks**

The enterprise information system architecture outlined above does not claim to be a complete solution to the development of virtual enterprise systems. Instead, its aim is to motivate the rest of the discussion and to provide the general framework for the problem this thesis seeks to address. The main premise of the discussion has been that a component-oriented development approach seems promising in providing the flexibility
and dynamism lacking from current enterprise information systems in order to fully realise the virtual enterprise vision. Following such an approach requires that an appropriate architecture be in place, upon which the enterprise system will be built. Such architecture in turn requires the development of a set of supporting services. Each one of these services in order to play its allocated role requires that some serious technical challenges be addressed. Taking into account the extent of the overall technical challenge of the outlined architecture it would have been unfeasible to attempt to address all services in this thesis. As a result and bearing in mind the instrumental role that the component location and selection service plays in the overall architecture, this service is the main motivation behind this thesis.

1.2. Problem statement

The aim of this thesis is twofold:

- To bring the notion of service location and discovery, widely used in large-scale networked and distributed systems, into the domain of component-based software development.
- In doing so, to also improve it by providing it at a higher quality than currently available.

In order to be clear on what exactly the aim is, the above goals need to be further elaborated on. So, when they are talking about the domain of component-based software development, they refer to this idea of developing systems that comprise of a number of software pieces, the components, composed together. Although the exact nature of these software components requires further investigation, it has already been shown in the previous section that they seem to be the only way to provide the high levels of flexibility and dynamism required in modern enterprise information systems. On the other hand, service location and discovery have been widely used to provide highly decoupled network and distributed systems that are flexible enough to handle failures or more generally changes in the services available to them. As a result, it seems reasonable to try to bring service location and discovery into the domain of component-based software development. We believe this to be a demanding problem because the move towards component-based software in general has proven quite challenging. Moreover, in the
light of some recent technological developments such undertaking seems also realistic. In this front, the measure for the success of this thesis is to demonstrate the feasibility of this idea.

When the aim refers to the improvement of current service location and discovery, the nature of the anticipated improvement needs to be explained. For this purpose the location and discovery process is considered as a retrieval process. Therefore, the typical information retrieval measures of precision, i.e. the percentage of retrieved items that are relevant, and recall, i.e. the percentage of relevant items retrieved, can be used to show improvement. To begin with it is not clear whether it is reasonable to expect such improvement. So, in this respect, this seems to be the most challenging part of the aim of this thesis.

1.3. Approach overview

In order to achieve the above aim first we need to find out, what is the nature of software components, and what the state-of-the-art component platforms currently offer. We carry out this investigation in Chapter 2. Its most important outcome is the identification of Definition 2.1 as a solid foundation for the rest of our approach.

Then in Chapter 3, we turn our attention in the area of service location and discovery in order to establish which particular approach would be the most appropriate to bring into the component-based development domain. Service trading is identified as the most promising one. We then analyse service trading form a component software point of view, in order to determine what are the main challenges we have to address to bring it into the component-based development domain. Our analysis singles out the syntactic nature of current service trading as its main shortcoming, and leads to the introduction of the notion of Semantically Enhanced Component Trading (SECT) as a new kind of trading that removes the weaknesses of service trading, and brings it into the component-based development domain. SECT achieves this, with the definition of a component type model that includes semantic notions of types and type relationships, i.e. types that incorporate explicit descriptions of type behaviour, which are utilised in establishing type relationships. In order to determine how to describe type behaviour, we turn our attention into the area of component retrieval in software reuse. There we
identify pre-/post-conditions, invariants, history constraints and model programs as necessary constructs for the description of type behaviour. We also identify approaches to signature and specification matching as candidates for incorporation in the definitions of type relationships. At the same time, we also examine work on object interoperability in order to establish which particular aspects of type behaviour are going to be included in the type descriptions. As a result, we create a list of defining characteristics for the component type model for SECT.

In Chapter 4, starting from these defining characteristics we specify all aspects of the component type model for SECT. This process starts with the identification of the kinds of types that are going to be included to the model. During this process:

- We introduce component types that provide and require sets of service types, and contextualised interfaces for component and service types.
- We introduce property types for the description of extra-functional aspects of service and component types, and incorporate the notion of extra-functional contract types to their definition.
- We adopt an object-based model for service behaviour and define a model of component behaviour based on interacting objects.

Subsequently our attention turns to the definition of a type naming scheme, that includes in addition to private and universal types, also standardised conformant types. Followed by the definition of rigorous type descriptions that can be utilised to define type relationships. Then, we focus on the type relationships of the model. At first though we set an underlying framework within which the type relationships can be defined. We define four kinds of type compatibility/conformance relationships:

- Name-based,
- Structure-based, which also incorporate a selection of signature matches,
- Syntactic-based, which combine name-based and structure-based ones,
- Semantics-based, which incorporate a selection of specification matches, primarily focusing on reuse-ensuring ones.

Based on the above kinds of type compatibility/conformance relationships, we then continue to define inheritance relationships that support inclusion polymorphism and can be used for incremental definition of types. The definition of the component type
model concludes with the introduction of type definition domains. These domains either support the definition of extra-functional types, or the definition of application domain specific concepts and types. The latter are also utilised in the definition of domain-enhanced name-based and syntactic-based type relationships.

In Chapter 5, we provide our evaluation of the SECT component type model. The evaluation is primarily as a foundation for component trading of higher quality compared to current service traders. It is carried out in two parts. First, we demonstrate how expressive the component type model is in precisely capturing differences between types, through a number of example type definitions. Second, we also explore the issues pertaining to the implementation of a SECT-enabled trader by providing a trader architecture that takes advantage of the particular features of the component type model. In general our evaluation shows that notion of SECT supported by the component type model has the potential to not only achieve its original aims, but may also be valuable in a wider context than the enterprise information systems that motivated its development.

We conclude this thesis in Chapter 6, where we identify the introduction of the notion of Semantically Enhanced Component Trading (SECT) that is solidly founded on a component type model incorporating notions of semantic types and type relationships as the primary original contribution of the thesis. In order to further clarify the original contributions of this thesis we carry out a comparison to closely related work. Before we present our main conclusions, which also examine the relevance of SECT in the context of web services and we find that it SECT is applicable in this domain. However, we also find that there is significant work based on semantic web technologies that could complement our approach. Our main conclusion is that SECT manages to bring service trading into the component-based development domain, with introduction of semantic notions of component types and component type relationships, and has the potential to enhance service trading both in terms of precision and recall, with the use of semantics-based and composite type relationships and the support of a composition facility. After that we identify three particular directions for future work: (a) extensions of the type model, primarily by introducing domain ontologies and more elaborate composite relationships, (b) realisation of trading architecture, particular with respect to supporting formal reasoning for UML/OCL expressions, and (c) refinement of the type model for
particular domains, in which case we explain what we mean by applying SECT to context-aware systems.
Chapter 2. Component software

In the previous chapter we described how the virtual enterprise model provided the motivation for this thesis. This description concluded with the identification of a component location and selection service as one of the core supporting services of a virtual enterprise information system architecture, and its selection as the main motivation of this thesis. In this and the following chapter we aim to put the problem of developing such a service in context by identifying related research areas and reviewing their literature. The main outcome of this review is the introduction of the notion of semantically enhanced component trading as our proposed solution to the component location and selection problem. This notion is in turn placed in context by reviewing additional related research that drive the development of its technical characteristics, elaborated in subsequent chapters.

More specifically, since the virtual enterprise information system architecture presented in section 1.1.2 is premised on the adoption of a component-based approach to software development, we devote this chapter to an investigation of component software. In recent years component software has attracted a lot of attention leading to quite extensive research literature in this area. Proving an in depth coverage of all this literature is outside the scope of this thesis. As a result, we only provide a brief overview in section 2.1. This overview also includes some pointers to the most interesting in our opinion sources of related material. Instead, our main intention is to establish exactly what software components are, what are their main technical characteristics and their implications. A detailed discussion of these issues is provided in section 2.2. Moreover, since in recent years component based software development has entered the mainstream through the introduction of supporting platforms by major industrial players like Microsoft, Sun Microsystems and OMG, we also review what is the current state of their offerings in sections 2.3.3, 2.3.1 and 2.3.2 respectively.

2.1. A brief overview of component software research

McIlroy in his seminal paper “Mass produced software components” [45] first introduced the idea of component based software development in the late sixties. The
idea was that instead of developing software systems every time from scratch, we compose them from prefabricated components in a process similar to the construction of hardware systems by plugging together circuit boards. This kind of change in the approach to software development is considered a necessity in order to tackle the software crisis, the fact that as time goes by we require more and more software systems of increasing complexity. Moreover, such a change will also help to transform software development from a high-risk activity, largely unpredictable and very costly, into a disciplined activity similar to traditional engineering, giving rise to software engineering.

The development of software systems from prefabricated components presents a number of technical and financial advantages [180, 84, 88, 101, 87]. First, the overall cost of developing the system is reduced since less software needs to be produced. Second, the risk of managing the software production process is considerably reduced as decreasing the amount of software development needed also reduces the scope for errors and misjudgements. Third, the development time is drastically cut, another consequence of the reduced needs for software development. Fourth, the trustworthiness of the produced system is significantly increased, because errors in tried and tested components are more likely to have been discovered and corrected. Finally, note that although component development itself may require significant effort and cost, the fact that the components are reused in a series of systems means that this effort and cost can be distributed over a number of projects, making not only feasible but also desirable the search for optimal solutions and implementations.

Since McIlroy first articulated his vision of turning software development into an engineering discipline, undoubtedly a lot of progress has been made. The increasing volume of software engineering literature is ample evidence of this. However, there is still some way to go. As is demonstrated in [46], software development techniques are still unable to provide the necessary productivity increases to keep up with the need for more and higher quality software. This failure can be attributed to the fact that although component based software development is now entering the mainstream, as evidenced by the introduction of component platforms from major industrial players like Microsoft and Sun; and software components are now commercially available, from vendors like
ComponentSource (www.ComponentSource.com), Flashline (www.flashline.com) and JARS (www.jars.com), it is still not the norm.

Over the years a number attempts have been made to introduce appropriate programming abstractions to support component software. The most notable of these are modules and objects, which led to modular [47] and object-oriented programming respectively. Both abstractions have been incorporated as core constructs in programming languages. For example modules are part of languages like Modula-2 and Ada, where they are called packages, while objects are part of a whole range of language from Simula67 and Smalltalk to C++ and Java. It is worth noting that objects have been considered as “better” modules [77] or in other words the next step in an evolutionary process from modules to components. In fact, for some time they were even considered to be the final step of this evolutionary process [48]. However, the realisation that object technology had in general failed to deliver in its promise for widespread code reuse combined with the observation that Visual Basic and ActiveX controls, the first commercially successful component technologies were not object-oriented contested this view [74]. It is now widely recognised that software components although similar are more than both modules and objects [7].

This realisation that component software requires more than modules and objects has spurred in recent years a lot of research activity. This activity at first was focused on identifying in which way component are different from modules and objects or in other words what modules and objects lack in order to fully fulfil the potential of component software. We explore this issue further in the following section. As these differences started to become clear, the research focus expanded to cover all aspects of component based software engineering. Research emphasis moved from issues of component functionality and interaction, to issues of quality, management, evolution, tools, and methodology (see [75] for a detailed list of questions and/or challenges on each of these aspects). All this research effort by both industry and academia has produced some significant results, mainly in the area of component models and supporting platforms for components further elaborated in section 2.3. However, despite all this effort Schneider and Han argue that “we still have not gotten the fundamentals of component-based Software Engineering right, as long as we cannot give a well-defined and generally-
accepted definition to the questions "what is a software component?" and "how do I correctly compose software components?" [75]. As a result, major challenges still remain in all aspects [76].

Since, our aim in this section is not to provide a complete and in depth analysis of component software research in general, we are concluding our introduction to component software with a brief presentation of the main sources of related published research. This presentation aims to act as an introductory guide to a research area that has grown extensively over the last decade.

One of the first researchers to focus on and to set the research agenda for a compositional approach to software development was O. Nierstrasz [83, 82 and 81]. However, this work was carried out at a time when the differences between object-orientation and component software were only starting to be appreciated, as also demonstrated by the title of the book, "Object-Oriented Software Composition" [55] edited by O. Nierstrasz and D. Tsichritzis and detailing the work of their research group in this area. It is also interesting to note that despite the progress in the area, the research agenda set by O. Nierstrasz still remains largely valid [75].

Certainly, one of the most prominent figures in this area is C. Szyperski (see his website at research.microsoft.com/users/csypers/). Starting in the mid nineties, his work on independent extensibility of systems [78]; the problems or deficiencies of objects from a components software perspective, or in other words, why objects are not enough [80] and the characteristics that software components should have [79], paved the way for a more systematic study of component software. He also set up with J. Bosch and W. Weck an international Workshop on Component Oriented Programming (WCOP) providing the first focal point for the research in this emerging area. It is interesting to note that the workshop is still running in 2004, providing one of the best sources for recent developments in the area. Moreover, he provided the first comprehensive overview of the related research in the first edition of "Component Software: Beyond Object Oriented Programming" [7]. In the second edition of the book [84], he provides an updated overview of the area where it becomes clear that not only significant progress has been made in certain respects, but also that component software nowadays encompasses
almost all aspects of software engineering. The book is certainly the best starting point for anyone interested in this area.

Another important source of related research is the Software Engineering Institute (SEI), a development and research centre sponsored by the U.S. Department of Defence and operated by Carnegie Mellon University (www.sei.cmu.edu). As component software started to mature and encompass more aspects of software engineering, SEI commenced a number of initiatives focusing on it. Of particular interest are:

- The COTS (Commercial-Of-The-Self) Based Systems (CBS) Initiative (www.sei.cmu.edu/cbs/), which focuses on "improving the techniques and practices used for assembling previously existing components into large software systems and migrating existing systems towards CBS approaches".

- Predictable Assembly from Certifiable Components (PACC) Initiative (www.sei.cmu.edu/pacc/), which focuses on "how component technology can be extended to achieve predictable assembly from certifiable components".

Both initiatives have produced a number of publications and technical reports most of which are available from the respective websites, while the much of the work in the former initiative has also been included in the book “Building Systems from Commercial Components” [85]. Moreover, SEI has supported a series of workshops and conferences in the area. Of particular interest for material relating to recent developments in the area are the International Symposium on Component-Based Software Engineering and the International Conference on COTS-Based software systems (www.iccbss.org).

It is also interesting to note that the interest of the PACC initiative in certifiable properties of components as the basis for trust in them is also shared by the Trusted Components (TC) Initiative (www.trusted-components.org). TC is "a cooperative effort to provide the software industry with methods, techniques and tools for building high-quality reusable components”. B. Meyer, C. Mingins and H. Schmidt commenced the initiative with their article “Providing trust components for the software industry” [86].

Further to the workshops and conferences mentioned above, another source of related material and recent developments is the Component-Based Software Engineering Track at the Euromicro Conference (www.idt.mdh.se/ecbse/). Moreover, the series of workshops on Specification and Verification of Component-Based Systems are focusing
more to the formal aspects of component software (www.cs.iastate.edu/~leavens/SAVCBS/), while the book “Foundations of Component-Based Systems” [2] provides a good introduction into this area. Finally, individual papers or even whole sessions about components software can also be found in the recent instances of the International Conference on Software Engineering (ICSE, www.icse-conferences.org), and reflecting its strong links with object oriented programming in the recent European Conferences on Object-Oriented Programming (ECOOP, www.ecoop.org) and the International Conferences on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA, www.oopsla.org).

2.2. What are software components?

As we have pointed out, according to Schneider and Han the research community is still looking for a consensus on what software components are [75]. As a result, there are a lot of proposed definitions to choose from. For example Szyperski lists fourteen different definitions (in chapter 11 of [84]) to which he adds three alternative forms of his own definition. Examining all these definitions reveals that there are widely different views of what are the basic characteristics that software components must have. Furthermore, even if there is some agreement on the basic characteristics, there are still differences on how these characteristics are emphasised in the proposed definitions. Recognising these differences and the fact that the notion of component is not well defined for practical purposes, Voelter has even proposed a taxonomy that captures the variety of features and characteristics [89]. Therefore, it is prudent that before any in depth technical discussion on component software that we examine these characteristics and identify the ones that form our notion of software components.

There is certainly agreement that components are units of composition and as such can be composed with other components to form a software system [87, p. 442]. However, this kind of definition is way too general and abstract for any practical purpose. It is interesting to note that this definition does not even preclude the case of traditional applications being considered as components. The reason for this is that this definition does not really require components to be parts of a composition.
In order to exclude traditional applications being considered components, Meyer requires that components are program elements that may be used by other program elements, their clients [88]. He also adds: “the clients and their authors do not need to be known to the [component’s] authors”. This latter property requires that components should be of interest to a broad range of clients that are not part of the particular software system for which the components were initially constructed [90]. In other words, it emphasises the fact that components should be reusable. Adopting reusability as the only core property of components results in a very wide view of what the program elements referred to in Meyer’s definition might really be. Consequently, depending on the process activity on which the components apply we can refer to analysis components, design components, etc [88].

Such an exclusive emphasis on reusability although in line with McIlroy’s vision is not widely accepted. Szyperski argues: “while reuse is indeed an important economic and technical issue, it’s not the main driver for component software” [91]. He, in fact, considers the main drivers to be the extensibility and evolvability of software systems without resulting in complete replacement of the existing system. As a result, he regards the core property of components to be their ability to be independently deployed [84, p. 36]. We should point out that reusability and independently deployment are in no way in conflict to each other. In general, the techniques that enable independent deployment also improve the reusability of components. Therefore, both of these properties are widely required from software components [87, 92], including Meyer [90] and Szyperski [84].

2.2.1. Software components: binary or not

The consensus, that components should be units of independent deployment, breaks when the implications are considered. According to one school of thought [84, 92, 87] independent deployment implies that components should be executable or binary in nature. The reason is that in order to extend and evolve existing systems without replacing them, we need to be able to load and install new components dynamically [91]. This means that the deployment of a component into the system is a largely automated process, i.e. requires minimal if at all human intervention. This can only be the case if the components are already in binary format, i.e. are targeting a particular execution
environment [95]. However, the opposing school of thought argues that the exact nature in this sense of components is not really important [94, 89, 93, 21].

At the root of the argument lies confusion regarding the meaning of terms binary and deployment. The rise of technologies like scripting languages and just in time compilers has blurred the distinction between executable and non-executable program elements [90]. Even if we focus on what is the target of the program element, humans and development tools or execution environments as in [95], this does not really make the distinction any clearer. For example scripting language have been developed to target both the execution environment containing their interpreter and humans. Moreover, the argument that if compilation is needed then there are too many dependencies for the component to be considered independent does not carry much weight either. This is the case since both sides of the argument agree that components can have dependencies on the deployment environment and even other components. As long as these dependencies are made clear and explicit, then the independence of the component is not compromise. Consequently, there is no reason why any compilation dependencies cannot be included.

There are of course limits on how many dependencies a component can have without losing its independence. Accordingly, we could say that the real difference between the two schools of thought is on where exactly this limit is placed. Another interesting point in this debate is that even if the components are not binary, we cannot really take advantage of this. We cannot for example inspect or change the available code, since this may compromise the reusability or independence of the component. As a result, adopting the wider view of components does not really buy us anything. In fact, we can even argue that considering non-binary components may be costly because it is more difficult to enforce information hiding, i.e. encapsulate the implementation details of the component, and to protect intellectual property by not revealing how the component really operates.

In conclusion, we don’t really believe that independent deployment implies binary form for components. However, we also believe that it is important to maintain the separation between the wide and narrow view of components, the real software components according to Szyperski [96], or as D’Souza and Wills put it between components in general from components in code [21], or between logical and technical
kinds of components [89]. First, because we want to separate components as applied to the deployment phase of the software process rather than to earlier phases. It is this kind of software components we consider in this thesis. Second, because we believe that thinking in terms of binary components emphasises encapsulation of their implementation details.

2.2.2. Software components: units of independent deployment and third party composition

So far we have identified three core characteristics of components: compositionality, the ability to be composed with other components to form software systems; reusability, the ability to be used as parts of software systems further than the one they were constructed for; and independent deployment, the ability to be easily integrated as a separate unit into an existing software system in order to extend or evolve its functionality. We should note that Szyperski condenses the first two of these characteristics into what he calls composition by third parties [84, p. 36]. In fact, the notion of composition by third parties is a bit wider as it also captures the market related considerations about software components. We will come back into these considerations later on. At the same time, Somerville using different terminology refers to similar characteristics properties when he talks about components being independent, composable and deployable [87, p. 443]. In addition to these properties he also talks about components being documented and standardised, but for the time being we do not consider these properties. However, using the above core characteristics it is still not very clear how we can determine if a software element can be characterised as a component or not.

To address this problem, Meyer expanded on his initial definition of software components [88] and identified seven specific criteria that can be used to determine if a software element is a software component or not. According to [90], these criteria are as follows:

1. *May be used by other software elements (clients).*
2. *May be used by clients without the intervention of the component’s developers.*
3. *Includes a specification of all dependencies* (hardware and software platform, versions, other components).

4. *Includes a precise specification of the functionalities it offers.*

5. *Is usable on the sole basis of that specification.*

6. *Is composable with other components.*

7. *Can be integrated into a system quickly and smoothly.*

It should be clear that these requirements map to the three core characteristics and aim to elaborate them. So, compositionality maps to first and the sixth criterion. The third, fourth and seventh criteria elaborate the independent deployment characteristic. Reusability is elaborated by the second and fifth. In fact, the fifth criterion is bit more general and refers to the need for information hiding, or in other words encapsulation of the component’s implementation. Information hiding promotes not only the reusability but also the independence of the component. Moreover, it should also be clear that the specification of offered functionalities and dependencies in a way that enforces information hiding is crucial for the characterisation of a software element as a component. How to achieve this is a major technical challenge. Finally, note that when Sommerville talks about components being documented — or in other words that this specification has to provide adequate information in order for potential users of a component to be able to decide if it is suitable or not — he refers to exactly this challenge.

A well-known approach for enforcing information hiding is by using interfaces. In this case the interface defines the encapsulation boundary. All the features provided by a software element that are available to its clients are described as part of its interface, while how these features are implemented is hidden behind it. Applying this well-established practice to software components, we can rephrase/replace the criteria three, four and five above with the following:

- Includes a description of the interface of its provided functionalities.
- Includes a description of the interface of the functionalities it depends on or requires.

This change leads us naturally to the definition of software components according to Szyperski (see Definition 2.1) [7, 84]. This definition is the one we adopt for the rest of this thesis. There are three reasons for this choice. First, the definition captures the core
characteristics of software components as discussed above. Second, it was the product of a consensus formed at the first International Workshop on Component-Oriented Programming. Third, it is by far the most widely referenced in the literature.

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

**Definition 2.1. Software Component.**

The discussion so far has covered most of the aspects of this definition. However, the definition does not just refer to interface as our above two points, it also requires that these interfaces are contractually specified. This is in order to highlight that the kind of interfaces required by software components are different from the traditional notion of interfaces popularised by Interface Definition Languages (IDLs). It is widely acknowledged that traditional interfaces describing a set of named operations and their arguments and return types are inadequate for the precise specification of the functionality provided by software components [90, 84, 93]. Meyer goes even further to the extent that he says: "IDLs as we know them today are doomed" [97]. What is in fact required is the specification of a detailed contract that binds the component itself and its clients and specifies what are the rights and obligations of either side [95, 84]. Such a contract is even more important when it also includes the context dependencies. Meyer describes such a contract from the point of view of a software component as consisting of three parts each answering one of the following questions: "What does it [the component] maintain? What does it expect? What does it guarantee?" [97] How to specify each one of these parts remains still largely a technical challenge [75, 96]. We will come back into this later on in this chapter.

2.2.3. **Software components: stateless or stateful**

Thus far, although we have adopted Szyperski’s definition of software components (see Definition 2.1), we have focused only on two out of the three characteristic properties of components he requires [84, p. 36], namely that a software component is a unit of independent deployment and a unit of third party composition. We
have purposely ignored that he also requires software component to have no (externally) observable state. The reason for this omission is that the question whether software components may or may not have state is an area of heated debate.

At first, we should point out that Szyperski initially required that software components have no persistent state [7, p. 30]. This observation bears the following questions: (a) what is the difference between the initial and the current form of the requirement; (b) what exactly does this requirement mean; (c) why is such a requirement needed? Regarding the first question, it is clear that the second form of the requirement is in fact stronger, i.e. more restrictive. While the initial form only prevented the preservation of state between successive activations of a component, the current form prevents any state dependent behaviour.

Regarding the second question, Szyperski clarifies that what is really important is that a component cannot be distinguished from copies of its own. However, he admits that there are some possible exceptions, for example attributes that do not contribute to the component’s functionality, like serial numbers used for accounting purposes; or technical uses of state that can be crucial for performance but do not affect observable behaviour, like using state for caching [84, p. 36]. In order to make the point clearer he also provides two examples: a database server component [84, p. 36-37] and a file system interface component [95]. In the case of the database server, the server with its database, e.g. a company’s payroll system, is not a component, since it can be seen as a module with observable state, i.e. it is one of potentially many instances. However, the static database server program is a component, since it is only a single instance. In the case of the file system interface, it can be a component given that it operates on file descriptors and as a result it does not need to maintain any state itself. The file descriptors encapsulate the state of its directories and files. In reality preventing components from having observable state enforces a separation of the immutable “plan”, the database server program or the file system interface, from the mutable “instances”, the database or the file system [98].

Regarding the third question, the brief answer is that it makes life easier by removing a lot of maintenance problems. Since, we cannot distinguish a component from copies of its own then it does not make sense to talk about a particular component
“instance”. As a result, components have a name, which is shared by all replicas that might exist, but not a unique identity that is a per-instance name [91]. Moreover, if there are multiple copies available, by having no observable state, all copies appear the same, i.e. cannot drift apart over time, and it does not make any difference which replica we use. On the other hand if components have state then no two installations would have the same properties [98]. As a result, clients would rely not to a particular component, but a particular component instance instead. In this case, replacement of components becomes more difficult, since it needs to take into consideration the particular state in which the component to be replaced is [96]. It is an interesting point that the two forms of the requirement relate to considerations about how dynamic component replacement can be. If we consider component replacement only at deployment time, then what complicates matters is persistent state. However, if we consider component replacement at runtime, i.e. dynamic replacement, then any state, whether persistent or not, complicates matters.

It is certainly the case that if components have no state, then dynamic replacement becomes easier, but it also raises the question how state dependent behaviour can be introduced in a component based system. For Szyperski the answer is in objects [98]. More specifically, “where a component needs to manipulate state, it should do so by means of objects. A component’s services should be invoked via methods of an object, where the component contains the class that implements these methods” [91]. So, different clients that depend on different state, would access the component through different objects encapsulating this state [95]. Following this line, “the differentiation of components and objects is about differentiating between static properties that hold for a particular configuration and dynamic properties of any particular computational scenario” [98]. According to this view, components are at the level of classes not objects and preventing from having observable state is like preventing classes from having behaviour dependent on static class attributes [99]. Although, in this sense the argument for preventing components from having observable state might sound quite reasonable, this supposes that ones Szyperski’s view on the role of component and objects in component based system. However, this view is by no means widely accepted.

An alternative and quite common view is to see components as collection of tightly coupled objects [93]. Following this view Douglass argues that behaviour in
general can be one of three possible categories: simple, history independent; continuous, history dependent but in a non-discrete way; or state, history dependent in a discrete way [93]. In all cases, a software element at any moment in time accepts a particular set of input events and can carry out a particular set of actions or activities. In the simple case the set of input events and actions is unchangeable over time, while in the non-simple cases it depends on the past history. Moreover, a software element can have any or all of the three behaviours in different behavioural aspects. As a result, components according to this view may have state [93].

The main advantage of the former view is that it simplifies the maintenance of component-based systems and makes the consideration of components as units of dynamic replacement and versioning more straightforward without though being a necessary condition. On the other hand, the second view makes the transition from and the exploitation of object-oriented technologies a lot more straightforward in the context of component based systems. For the time being there is certainly not enough evidence available to suggest that either view is more appropriate. However, it is interesting to note that component models like OMG’s CCM and Sun’s Enterprise JavaBeans (see the following section for details) support stateful components. For this reason we take the same view as Sommerville [87, p. 442] and we do not prevent components from having observable state.

2.2.4. Software components: component models, component architecture and standards

In the discussion about the core properties of software components we identified compositionality as one of them. We should though point out that unless the components adhere to some predefined constraints or conventions their composition is not possible [101]. These constraints and conventions are usually defined at two separate levels: the component model and the component architecture.

The component model usually describes conventions relating to interfaces, the use of components programmatically and their deployment [87, p.446]. The elements relating to interfaces describe how the component contracts, their provided and required interfaces, should be specified and how the components can be composed, i.e. the
supported composition approaches. Each component platform is usually targeted towards a particular component model, in some cases it may even target multiple component models; and provides a supporting environment for component composition, in other words guarantees that components targeting this platform can at least interact with each other. It is worth noting that current component platforms (see following section for the description of the most prominent ones) support mainly two composition approaches: connection-oriented, where a component can be directly connected to other components that provide the interfaces it requires; and container based, or contextual composition, where components are placed into a container which defines their context and manages their interaction with other components [102].

The fact that a set of components has been developed for the same platform even though it guarantees that they can interact with each other, it does not guarantee that these components will form a coherent composition. Unless these components have been designed following the same set of guidelines aiming to achieve the specific goals of the composition, or in other words have been targeted for the same architecture, it is quite likely that significant problems will compromise the composition [103]. This is the reason why according to Szyperski besides targeting a specific platform, software components also assume architectural embedding [96].

It is important to note that the constraints and conventions introduced by component models and architectures unless fairly common can be quite restrictive and may seriously undermine the reusability of the components. This is the reason why both Szyperski and Sommerville emphasise the importance of standards [84, 87]. In the case of component models and their supporting platforms standardisation is clearly the norm, although it may not follow the traditional routes of international standardisation bodies, using community or industry based standardisation procedures. On the other hand, component architectures are still lacking in this respect. The main reasons are that first it does not make sense to talk about standardised component architectures before the existence of standardised component platform, which only recently have emerged. Second, component architecture standardisation does not make sense across the board, since different application domains have different needs. This requires efforts targeting particular application domains, which only have started to emerge and are still to deliver.
It is our belief that as domain standards start to emerge, the standardisation would not be limited only to component architectures but it would also embrace components themselves. This will draw attention to a particular type of software component, namely business components. Herzum and Sims define business components as: "the software implementation of an autonomous business concept or business process. It consists of all the software artefacts necessary to represent, implement, and deploy a given business concept as an autonomous, reusable element of a larger distributed information system" [100]. We consider that such an evolutionary standardisation process is the only way to control the complexity of software component development. We should note that when talking about standardisation of business components, we refer to standardisation of their characteristics in terms of provided and required functionalities.

2.2.5. COTS components

When we introduced the definition for component software (Definition 2.1), we said that the requirement for composition by third parties aims to capture also market related considerations without though elaborating on this. The reason why Szyperski considers market related issues is because there is a wide belief that the full benefits of component based software development will only be realised when an open market of software components is available. According to this view, when considering software components we cannot ignore the effect of market forces. In fact, Wallnau et al. claim: "market forces play as much a part in software engineering as friction plays in mechanical engineering" [92]. The examination of the issues relating to component based software development from an open component market perspective gave rise to the notion Commercial Of The Self (COTS) components.

In general, there is a lot of confusion regarding the meaning of COTS [104, 105] and to make matters worse there is also a plethora of other seemingly related acronyms (see [104] for an enumeration). However, it is clear that the term is quite general and does not only refer to software components. According to Morisio and Torchiano, "COTS products and components are two sets with a non-empty intersection" [105]. There also seems to be consensus that there are two particular aspects in which COTS products and subsequently COTS components, are distinct from other types of software, namely source
and modification/customisation [104, 105]. The former refers to the fact that the development and production of COTS products is controlled by a different organisation than the one that is using them as part of a software system. Moreover, the COTS products or components aim to address the needs of more than a few software systems. The latter refers to the fact that the ability to change the COTS product or component in order to fit the needs of the software system in which it is used, is quite limited, quite often restricted to simple parameterisation. Following a similar thinking to what we presented in the discussion on the binary nature of software components in general, this latter aspect is usually expressed as a requirement for the buyer to have no access to the source code (see for example [106]). However, in the case of COTS components the protection of the vendor’s intellectual property plays a more prominent role in the argument.

These distinct characteristics of COTS components have significant implications. The limited modifiability of these components coupled with issues like vendor relationships, licence administration, and training and cultural transition require a change in both technical and management processes in order to address COTS based software (CBS) issues and opportunities [106]. As a result, a lot of the literature on CBS is focused on software development methodology and management issues [110, 85]. This is particularly the case, since it is recognised that despite the fact that many systems have used COTS successfully for cost reduction and early delivery, CBS is still largely a higher risk activity than non-CBS software [106].

In CBS development methodologies particular emphasis is placed on a component selection phase, which given a large repository of available COTS components, aims to select the appropriate set on which the development of the system will be based upon [87, 85]. This selection process may seem very similar to the component selection process we described in our motivation (see section 1.1.2). However, there are some important differences. First, the CBS selection process targets commercial available components. As a result, it places a lot of emphasis on the description of the commercial aspects of these components, e.g. vendor information, licensing and pricing information, etc. In this thesis, we are not particularly concerned with this type of information. Still we are going to show how we can complement our approach with this type of information. Second, in
contrast to our approach that focuses more on the functionalities provided and required by components, the CBS selection process focuses on balancing a number of competing forces [109]. These forces are: the requirements of system under development, the provision of the available COTS components, project management considerations like costs and schedules, the overall risk of the development, etc. Considering the complexity of this balancing task, the CBS selection process tends to focus on quite simplistic descriptions of the provided and required functionality of components (see [107, 108] for example). The more in depth assessment of the suitability of the selected components is left for a later stage.

Finally, again because of the limited modifiability of COTS components, architectural incompatibility is a serious problem. In fact, it is claimed that because of these incompatibilities the development effort for CBS software scales with the square of the number of independently developed COTS components used [106]. Therefore, in CBS standardised domain architectures are even more crucial, because they offer their only way of keeping this problem under control.

2.3. Component platforms: current state

In the discussion so far, we presented mainly the academic perspective on software components. From this presentation it should be clear that component software has attracted significant research interest, and despite the fact that a number of issues are still open, considerable progress has been made. In this section we turn our attention to the industry’s perspective. The two perspectives are in some sense complementary, because while the academics aim to address the fundamental issues and to discover the basic tools and techniques that will fully realise McIlroy’s vision; industry takes a more pragmatic view and tries to utilise research results in practical ways. For this reason, industry’s and academia’s views of what software components are and how component based software development should be carried out differ in a number of respects. Quite often particular techniques and tools that academia advocates are not considered mature and practical enough to qualify for industrial use.

The industrial perspective is expressed mainly in two ways either through the various products that companies provide or through various industry supported standards.
As it would be expected different products and standards tend to take quite different views on the various issues. Within the industrial world three different perspectives on component software are currently dominant: the Java perspective, presented by a number of related technologies introduced by Sun Microsystems and a supporting user community; the Object Management Group (OMG) perspective, presented by a number of industry de facto standards supported by one of the largest industry consortia; and the Microsoft perspective, presented by a number of technology and standard introduced by Microsoft. The dominance of the three perspectives is evident by the large numbers of organisations adopting the proposed technologies and standards. Moreover, in section 1.1.2, we identified four technology trends that make McIlroy’s vision appear today more feasible than ever before, namely the Internet and the World Wide Web, machine-independent language and object technology and compositional elements in software products. It is interesting to note that all three dominant industrial perspectives adopt all four of these technology trends in their own particular form.

More specifically, Java started as an object-oriented machine-independent language. It then targeted the World Wide Web with technologies like applets and servlets making web pages dynamic; whereas, it introduced compositional elements initially in web pages and nowadays in any application with JavaBeans and Enterprise JavaBeans. OMG started with a platform independent architecture for distributed object systems, which gradually evolved into an attempt to provide an enhanced and programming language independent version of the Enterprise JavaBeans model. Microsoft started from compositional elements in the form of VBX and ActiveX components and OLE, and it gradually moved to a more general component model (COM) with added distribution capabilities (DCOM). At the same time, it targeted the World Wide Web with technologies such as Active Server Pages and ODBC. Finally, it brought all its technologies together into a single framework (.NET) adding a new machine-independent language (C#).

In the following sections we examine briefly each one of these three perspectives.
2.3.1. The Java perspective: JavaBeans and Enterprise JavaBeans

Sun Microsystems introduced the Java language and platform in the mid-1990s. The machine independent nature of the language and an emphasis on the Web were the main reasons for its success. Java is able to abstract away from the underlying platform by using a Virtual Machine (JVM). The idea is that programs are compiled into bytecode, which can be executed unchanged in any machine that runs a JVM. The Java platform usually includes a runtime environment, which is an implementation of the JVM and the platform edition API, and a Software Development Kit (SDK), which includes tools like a compiler, a debugger and a documentation production tool javadoc. There are currently three platform editions: the micro edition, aimed at small and embedded devices; the standard edition, aimed at desktop computers; and the enterprise edition, aimed at multi-tier enterprise systems.

![J2EE architectural overview](image)

**Figure 2.1. J2EE architectural overview.**

From the three the most interesting from a component software point of view is the enterprise edition, which defines three groups of component models. Each of these groups targets a different tier of the typical multi-tier enterprise system architecture (see Figure 2.1). More specifically, on the client side there are: application components,

---

1 Figure 2.1 is a reproduction of Figure 14.3 from [84, p. 269].
JavaBeans and applets. On the web server tier there are: servlets and JSP. While on the application server tier there are: Enterprise JavaBeans. In reality, although JavaBeans are most often used at the user interface level, they could be used at any of the tiers. Java facilitates the interaction between the tiers and the various component models through supporting services, like naming, directory and messaging. Naming and directory services are accessible via a uniform API provided by the Java Naming and Directory Interface (JNDI). Messaging systems are accessible via the Java Message Service (JMS), which supports message queues for point-to-point message delivery and message topics that allow for multiple targets to subscribe. It also supports a variety of message types and the setting of message filters. In addition to these, Java also defines APIs for access to databases, the Java Database Connectivity (JDBC), and to other enterprise information systems, like enterprise resource planning, enterprise asset management and customer relationship management systems, through the Java Connector Architecture (JCA).

**The Java language**

Before, looking into each of the Java component models in more detail, we need to present briefly the most interesting features of the Java language itself and its basic services. Java is an almost pure object oriented language, which supports three kinds of types:

- Objects, which are either instances of classes or arrays
- Interfaces
- Primitive types like boolean, byte, char, short, int, long, float and double

It also supports single class inheritance, where the child class inherits also the code of its parent, and multiple interface inheritance. At the same time, classes may implement any number of interfaces. Moreover, in Java all classes and interfaces belong to packages, which introduce another level of encapsulation on top of classes. Packages form a hierarchy, which introduces a structured namespace for class features. Two interesting features of Java are exception handling and garbage collection. Exceptions can be used to indicate particular runtime errors, and are thrown either explicitly when an object detects an error or by the Java runtime. Garbage collection involves the removal of any objects that are no longer accessible, i.e. referenced.
It is interesting to note that because Java allows multiple interface inheritance but
only single class inheritance it prevents the diamond inheritance problem. This problem
occurs when a class inherits from two other classes, which in turn inherit from the same
class. In this case, for the methods inherited from the common parent class, it is not clear
which implementation to use. However, Java still has problems when multiple interfaces
define methods with the same name even if they have different signatures. For this
reason, it does not allow classes to implement such interfaces. At the same time, the
restriction of class inheritance to a single class can prove awkward in situations where a
class needs to implement functionality defined in two unrelated classes. In order to find
out which interfaces a class implements Java provides a type test operator,
instanceof, checked type casts and the core reflection service. Besides introspection,
the reflection service also supports modification of class, object and array fields,
construction of object and arrays, and method invocations.

Beside the reflection service, two other interesting basic Java services are object
serialisation and the Java Native Interface (JNI). The former defines a standard serial
encoding scheme and provides the mechanisms to code and decode webs of objects. It
creates a stream of bytes in a single pass process, thus allowing the part of the stream that
has been produced already to be forwarded to the destination. The stream is fully self-
described down to the level of Java primitive types. For an object to be serialisable, it
needs to implement the interface java.io.Serializable. Fields referring to
serialisable objects are also automatically serialised, while shared references to objects
are preserved. An alternative approach is to implement the interface
Externalizable, in which case the object’s fields instead of being automatically
handled, are left up to the object itself to handle. On the other hand, the JNI specifies for
each platform the native calling conventions when interfacing to native code outside the
JVM. It also specifies how such external code can access Java objects for which
references where passed, thus allowing external methods to: create, inspect and update
Java objects; call Java methods; catch and throw exceptions; load classes and obtain class
information; and perform runtime type checking. At runtime JNI uses a structure
identical to COM, without though being automatically compatible with it. The JNI
interface does not include standard COM functions, like QueryInterface.
In addition to the basic service, Java also supports a distributed object model through its Remote Method Invocation (RMI) interface. Interfaces that can be accessed remotely have to be derived from `java.rmi.Remote`. Remote operation can always fail as a result of network or remote hardware problems. As a result, all such methods are required to declare the exception `java.rmi.RemoteException`. If a method argument is of a remote interface type then the reference will be passed, while in all other cases arguments are passed by value using the serialisation service. We should also point out that Java garbage collection is fully distributed, taking into account both local and remote references to objects.

The Java component models

As we mentioned above Java defines five different component models: applets, JavaBeans, Enterprise JavaBeans, Servlets, and application client components. The first two are part of the standard platform edition, while the rest are part of the enterprise platform edition. This variety of supported component models reflects different developer needs at different enterprise architecture tiers, but at the same time introduces a lot of market fragmentation. As a result nowadays only a few these kind of components are used beyond the one enterprise application for which they were developed.

The first Java component model was applets, aiming at downloadable lightweight components that would augment websites displayed in a browser. The applet composition model is rudimentary, as applets can only be composed into a web page, where they can refer to other applets of the same page by name. However, there is no way to guarantee that a referred applet is of a particular type.

The second Java component model, JavaBeans, focuses on supporting connection-oriented programming and is as such useful on both clients and servers. Beans, as they are more commonly known, can be used to implement controls similar to OCX or ActiveX, and have been designed with a dual model in mind. At development time tools can be used to find out and configure the properties of beans and to connect them with other beans. At runtime the bean can use the properties to customise its appearance and behaviour. So, the main aspects of the bean model are the following:

- Events. Beans can announce that their instances are potential sources or listeners of specific types of events. An assembly tool can then connect sources and
listeners. Events are objects created by an event source and propagated to all currently registered listeners. Event-based communication can either be multicast or unicast. The way event communication is handled is similar to COM and to the observer pattern [114]. An event object should not have public fields and should usually be considered immutable. The model also introduces simple binary meta-attribution in the form of marker interfaces for the definition of event listeners. Marker interfaces are empty interfaces that mark the type that implements them.

- **Properties.** Beans expose a set of properties of arbitrary types by means of pairs of getter and setter methods. Note that JavaBeans use a combination of rules for the formation of the getter and setter method signature including its return type and its name. Szyperski refers to these rules as method patterns [84, p. 289-290]. The bean properties are named attributes that can affect a bean instance's appearance or behaviour. Indexed properties are also supported for cases where an array of property values is needed. Bound property changes trigger property change events, while constrained properties can only be changed if the change is not vetoed.

- **Introspection.** An assembly tool can inspect a bean to find out about the properties, events, and methods that it bean supports. This mechanism allows also attachment of arbitrary named custom attributes, i.e. name value pairs, to features of a bean.

- **Customisation.** Using assembly tools a bean instance may be customised by setting its properties.

- **Persistence.** Customised and connected beans instances need to be saved for reloading at the time of application use.

An interesting feature of JavaBeans is the introduction of Java Archive (JAR) files, which are used to package JavaBeans. Technically they are ZIP-format archive files that include a manifest file. The manifest file is used to provide information about the contents of the archive. A JAR file may include: a set of class files; a set of serialised objects that is often used for bean prototype instances; optional help files in HTML; optional localisation information used by the bean to localise itself; optional icons; and
other resources needed by the bean. There can be multiple beans in a single JAR file. In this case, the manifest file can be used to name the beans.

At the same time, an interesting extension to the JavaBeans model is the introduction of InfoBus. The InfoBus specification creates a generic framework for data-driven composition. According to the specification, beans are designed to be InfoBus-aware and are categorised as data producers, data consumers, or data controllers; all of which are coupled by an information bus, which determines data flow.

Moving to the enterprise platform edition component models, it is worth noting that all components of these models are packaged into JAR files, which can be included in an application. For distribution and deployment purposes, Java enterprise platform edition applications are packaged in .ear files that contain .war files packaging servlets and JSPs and .jar files packaging applets, JavaBeans, and Enterprise JavaBeans (EJBs). Moreover, all these components support the use of deployment descriptors, which are XML files co-packaged with a component describing how a particular component should be deployed. They serve two purposes: they enable the component developer to communicate requirements on the side of the component; and they enable the component deployer to fill in the blanks. Note that in COM+ and CLR these two purposes are kept separate. The developer’s information is captured in attributes aligned with the code, while the deployer’s information is kept in separate XML based configuration files. As it would be expected, the detailed nature of the deployment descriptors depends on the particular component model.

The third Java component model is servlets, which extended the basic idea of applets to the server side. They are usually lightweight components instantiated by a server inside a servlet container during processing of, typically, web pages. The servlet container supports its servlets by providing access to HTTP request parameters and by managing sessions. Moreover, servlets can also build on the services of EJBs. In fact, it is quite common to collocate servlet and EJB containers on the same machine, or even in the same JVM instance. A matching technology, Java Server Pages can be used to declaratively define web pages to be generated. During processing JSPs are compiled into servlets. In order to keep JSP pages largely contents-oriented, it is useful to minimise the code contained in JSP pages. For this purpose, JSP introduces tags, which can be used to
eliminate server side code in JSP pages. The JSP standard tag library (JSTL) includes tags for control flow, iteration, conditional, creation and configuration of JavaBeans, manipulation of XML documents, SQL, etc. Note that servlets inverse the web page programming model, while in JSP pages, code fragments reside in line with HTML, in servlets, HTML resides in line with Java source code.

The fourth java component model is Enterprise JavaBeans, which focuses on container-integrated services supporting EJBs that request services using declarative attributes and deployment descriptors. In other words, EJB is particularly geared towards contextual composition, which is the automatic composition of components with appropriate services and resources. More recently, with the addition of message-driven EJBs, it also provides support for data-driven composition. However, despite its similar name to JavaBean, EJB has no provision for connection-oriented programming. The addition of this is the main improvement of the CORBA Component Model to EJB.

The way contextual composition works is by placing a hull, a wall of proxies, around components and intercepting all communication from and to them. The components interact with services through a reference to their context. The combination of service implementations, intercepting hull, and context is referred to as a container provided by a server. In EJB the hull is realised by restricting all access to the features of EJBs to use two interfaces: a home interface for lifecycle management operations and the object interface for all methods of the EJBs themselves. Java's remote method invocation (RMI) is used to provide access to these two interfaces to non-local clients.

There are four kinds of EJBs: stateless session, stateful session, entity and message-driven beans. In all cases the home interface has a standard method to instantiate EJBs, while in the case of entity beans it also has a standard method to locate an existing bean by its primary key. The EJB container cycles beans through defined lifetime stages and serialises all invocations of beans, so there is no need to synchronise within beans. Moreover, the container protects beans from re-entrant calls throwing an exception whenever such calls are attempted and isolates faults.

More specifically, session beans are session-specific contact points for clients. Stateless session beans, as the name implies, do not maintain state across invocations, while stateful sessions can retain state across method invocations. Session beans have an
option to either explicitly control transactions, so-called bean managed transactions, or follow the default of container-managed transactions. At the same time, entity beans are supposed to correspond to database entities and encapsulate access to actual database records. For entity beans, the container is responsible to assign them primary keys, which can subsequently be used to locate the beans. The mapping of entity beans to and from database entities is called persistence and can either be bean-managed by directly using JDBC, or container-managed driven by an object-to-table mapping defined during the deployment process. Moreover, relationships between database entities define relationships between entity beans that can also be either bean or container managed. In general, EJB support one-to-one, one-to-many and many-to-many relationships, all in both unidirectional and bi-directional versions. In addition it introduces a query language similar to SQL called EJB QL, which is used in deployment descriptors to declaratively specify additional find methods for the home interface of entity beans. On the other hand, message driven beans have neither a remote nor a local object interface, nor a home interface. The only way to instantiate them and use them is to register them for a particular message queue or topic of the JMS. As a result, their context includes a JMS provider. Message driven beans when handling messages behave like stateless session beans handling incoming method calls, but unlike other kind of beans, they cannot return results. Finally, they do not have persistent state.

The fifth Java component model is the application client components, which are essentially unconstrained Java applications that reside on clients.

As a final point, we should mention that Java relates to CORBA in a number of ways. First, Java is one of the programming languages supported by CORBA. Consequently, a mapping from OMG IDL to Java has been defined. However, in contrast to other languages a mapping from Java to OMG IDL, usually referred as Java IDL, has also been defined. Second, besides the normal implementation of RMI using a proprietary protocol, an RMI-over-IIOP specification has also been introduced. This specification supports a restricted RMI variant, which enables RMI calls to reach CORBA compliant ORBs. However, this specification does not support distributed garbage collection. Finally, the Java Transaction Service (JTS) is a Java implementation of the CORBA
object transaction service. EJB introduced a much simpler interface to the JTS, the Java transaction API.

2.3.2. The OMG perspective: CORBA and CCM

The OMG perspective is built on top of the Common Object Request Broker Architecture (CORBA). CORBA was initially devised as a standard enabling collaboration of objects over the network in a language and platform independent way, i.e. as a wiring standard. However, it soon became part of a more general architecture that introduced a number of services and facilities. Furthermore, in its most recent version, CORBA 3, it has also added a component model making it a complete component platform. OMG, in general, takes a quite open approach allowing implementations of the standard to add value by introducing additional features. The openness of the approach means that in order to guarantee interoperability between different implementations a high-level protocol is needed. The most prominent such protocol is the Internet Inter-ORB Protocol (IIOP).

More specifically, at the centre of CORBA is the Object Request Broker (ORB). Servers that want to make their objects available over the network have to register them with the ORB using an object adapter. The main function of the object adapter is to mediate between the ORB and the actual implementation of the objects, called object servants. As part of this role, the object adapter creates object references; it is responsible for informing the ORB about which servants implement which objects, and for providing services for the activation and de-activation of these servants. In general, an object adapter can service any number of servants and there are a number of different policies for the management of servants. For this reason, object references include both the name of the object adapter, provided by the ORB and the name of the particular servant, provided by the object adapter. In fact, the ORB maintains a registry of both object adapters and servants, called the implementation repository, and is capable of loading and starting object servants when receiving invocation requests for an object that they serve. The use of the ORB allows clients, as long as they have an object reference, to interact with the object servant, pointed to by this reference, without really knowing where in the network it resides.
In addition to location transparency CORBA also provides distribution transparency to interacting objects. This is achieved by using stubs and skeletons at the client and server side respectively. Both stub and skeletons provide a local proxy [114] that deals with the marshalling and unmarshalling of invocation parameters and results. More specifically (see Figure 2.2), the client stub marshals the invocation name and parameters, which are then forwarded together with the object reference to the ORB. The ORB uses the object reference to locate the object adapter of the server and to forward the invocation information to the appropriate servant skeleton. The skeleton unmarshals the parameters and passes them to the servant that carries out the invocation. If the invocation produces a return result, then this is forwarded to the skeleton, which marshals it and forwards it to the client stub. Finally, the client stub unmarshals the received result and passes it to the client.

![Figure 2.2. Client/Server interaction in CORBA.](image)

For the client stub and the server skeleton to work all object interfaces need to be described in a shared language and the marshalling and unmarshalling mechanisms need to target this language. In order to enable the use of generic marshalling and unmarshalling mechanisms CORBA defines a common language for this purpose the OMG Interface Definition Language (IDL). At this point, we should also note that
despite their similar names Microsoft IDL and Java IDL refer to a different interface definition language and an OMG IDL to Java mapping respectively. So, in this section, whenever we use the term IDL, we refer to the OMG IDL. Since CORBA is language independent, all languages used must have bindings to the common language. This allows calls from or to a particular language to be related to the common language. As a result, all CORBA implementation can use IDL compilers, which compile IDL descriptions, and produce the necessary stubs and skeletons, and even outline code for the implementation of the servants. These IDL descriptions are also deposited in an interface repository, which every ORB is required to have. This makes them available at runtime for introspection purposes.

IDL distinguishes between data types and CORBA object references, which are distinct from usual object reference of the programming language used. The ORB provides operations to convert CORBA references into usual object references and back, as well as for converting them into strings and back. These strings are commonly used to store the CORBA object references. The data types are divided into basic and constructed ones, and include integers, floats, characters, strings, structures, sequences and multidimensional fixed size arrays. In CORBA, all data types are passed by value. Object may also be passed by value.

Besides the static selection of operations to be invoked described by IDL at compile time, CORBA also provides a Dynamic Invocation Interface (DII) and a Dynamic Skeleton Interface (DSI). These interfaces allow operation selection at runtime at the client and server side respectively. They both utilise a universal structure for the invocation and its parameters.

It is interesting to note that almost all CORBA standards are specified over interfaces defined in IDL. In CORBA 3 two new definition languages are introduced: the Persistent State Definition Language (PSDL), which captures storage types and storage homes; and the Component Implementation Definition Language (CIDL), which extends PSDL, adding components, component homes, composition entities, composition processes and executors.

The Object Management Architecture (OMA) adds to CORBA, (i) a set of common object service specifications, called CORBA services, (ii) a set of common
facility specifications, called CORBA facilities and (iii) a set of application object specifications, while CORBA 3 also adds the CORBA Component Model (CCM). Object services support all CORBA based programs in a way that is independent of application domains or models. Common facilities are either horizontal, i.e. domain independent focusing on specific application models, or vertical, i.e. domain specific. Both vertical and horizontal facilities define a component framework that can be used to integrate components. The former are defined by domain task forces, currently these include business enterprise integration; command, control, computer, communications and intelligence; finance; healthcare; life sciences research; manufacturing; space; telecommunications; transportation; and utilities. Application objects add domain specific entities that could be incorporated into component frameworks. The most prominent class of application objects is business objects, which are objects that directly represent abstractions used in specific businesses. Although application objects have been an area of great interest for years, the standardisation process in this space has been very slow. Despite several successful examples of application object models, the time is probably not yet right for general standards.

CORBA services includes 16 object services, one of which (the notification service) is formally part of the telecommunications domain facility. These service are divided by Siegel [115] into two categories: services relevant to today’s enterprise applications using CORBA, which typically use CORBA objects as modules and CORBA as a convenient communication middleware and these services supporting large scale operations; and services aiming at finer grained use of objects. The former category includes services like naming, trading, event, notification, object transaction and security. The latter category includes services like concurrency control, licensing, lifecycle, relationship persistent state, externalisation, properties, time, object query, and object collections. All these services have not been equally successful. In fact, for some of them there are no commercial implementations available.

The CORBA component model

The CORBA Component Model (CCM) is basically an extension of Sun’s Enterprise JavaBeans. According, to the model applications are assemblies of components, possibly combining both EJB and CCM components. These components are
shipped in packages containing the binaries of the components and an XML document describing other packages that these refer to, as well as their deployment configuration, i.e. a deployment descriptor. Moreover, these components can be divided into segments, which are the units of independent loading. CCM supports both component aware and component unaware clients. The former can take advantage of the full functionality of the model, while the latter cannot take advantage of features like the navigation between operations.

CCM classifies components into four categories: service, session, entity and process. The first three of these categories correspond to the stateless and stateful session beans and entity beans of EJB respectively. The category of a component determines how the object adapter treats component instances. Service components are instantiated per incoming call and do not maintain state across calls. Session components maintain state for the duration of a transactional session, but not across sessions. They also allow multiple calls within such a session. Process components have arbitrary lifetime, which corresponds to the business process they support. They also have persistent state. Entity components have persistent state and usually correspond to entities in some database. They are also assigned a primary key, which corresponds to the database entity’s primary key.

According to CCM, each component has the following features:

- A set of ports. These ports are classified into facets, provided interfaces; receptacles, required interfaces; event sources; and event sinks. During instantiation, a component’s receptacles need to be connected to facets of other components, while its event sources and sinks need to be connected to appropriate event channels. These connections can be either made declaratively at deployment configurations, or made dynamically at runtime. A special facet of all components is the equivalent interface, which enables navigation between the different facets of a CCM component. The support of multiple facets and navigation between them via the equivalence interface is very similar to COM and its IUnknown interface.

- A primary key. This is used for the identification instances in the case of entity components.
• A set of attributes and configuration interfaces. Attributes are named values exposed via accessors and mutators. Configuration interfaces are also described as IDL attributes with set and get operations. However, they only support initial configuration of new instances. Note that in CORBA 3, IDL is extended to allow exceptions to be thrown by these setters and getters. Moreover, a special call signals completion of configuration. Before this signal calls on operational interfaces are not allowed, while after it calls on configuration interfaces are not allowed.

• A set of home interfaces. These interfaces provide factory functionality to create new instances. They also have other lifecycle related operations for the objects they manage. Note that a component can provide multiple home interfaces.

At runtime, every component instance is placed inside containers, which provide a number of interface that support interaction with the object adapter, transactions, security, persistence and notification services. Containers also provide receptacles for the acceptance of callbacks to the component instance. The transaction control can either be container-managed or self-managed. In the former case, the component configuration states if transactions are supported, required, required new, or not supported. Similarly, persistence can be declared as either container-managed or self-managed. In the former case, PSDL is used to declare what needs to be persisted. For security management, the required access permissions can be declared on operations in CIDL. It is interesting to point out that CORBA 3 defines a component implementation framework, which includes generators that accept CIDL input and generate implementation code that completes the explicitly provided component code.

As part of its overall architecture OMG has also specified a Meta-Object Facility (MOF), which organises descriptive information in four layers. These layers are:

• The instance layer M0, which contains regular runtime instances;
• The model layer M1, which contains model types, the instances of which can be found at M0;
• The meta-model layer M2, which contains the entities of the modelling language, e.g. UML, used at M1. Note that the CCM introduces two new MOF M2 meta-models for OMG IDL and for CIDL.

• The meta-meta-model layer M3, which contains the MOF modelling entities used to describe the model of the modelling language at M2. For this purpose the fixed MOF core is defined, which is a subset of the UML core. OMG also defines a standard for mapping MOF meta-information to XML documents, called XML Metadata Interchange (XMI) [34].

2.3.3. The Microsoft perspective: VBX, ActiveX, OLE, COM, COM+ and .NET

Microsoft follows an evolutionary approach in the development of its component platform. This approach started with Visual Basic controls, which were the first commercially successful component technology and has gradually evolved the .NET framework. Looking at this evolutionary process there are two distinct phases. The first was built around the Component Object Model (COM) [72] and the second, which is currently still under development, is built around the Common Language Interface (CLI) [111] and is known as the .NET framework [66]. The fact that ActiveX controls [71], arguably the most successful component technology, is COM based combined with the

\[ Figure 2.3. The Structure of COM components\]².

² Figure 2.3 is a reproduction of Figure 15.1 from [84, p. 331].
fact that .NET is still in its early days and incorporates a number of COM related technologies, means that both phases need to be examined.

The first phase: COM, VBX, ActiveX, OLE and COM+

We start our presentation with COM. The fundamental elements in COM are interfaces, and being a binary standard it specifies that interfaces, at the binary level, should be represented as a pointer to an interface node. The interface node is in turn specified to hold in its first field another pointer. This second pointer is specified to point to a table of function pointers (see Figure 2.3). It is interesting to note that despite its name COM does not specify what components or objects are. Moreover, it does not require the use of objects to implement components. A COM component can implement several COM classes, each identified by a class ID (CLSID), which is a GUID, a 128-bit number guaranteed to be globally unique. Each COM class can implement any number of interfaces, which are named using interface identifiers (IID), which are also GUIDs. For the sake of convenience they may also have readable name, but these names are not guaranteed to be unique. COM objects are instances of a COM class, but do not necessarily constitute a single object (see Figure 2.4).

Figure 2.4. COM Component with multiple interfaces and objects.

---

3 Figure 2.4 is a reproduction of Figure 15.2 from [84, p. 332].
COM specifies that within a COM object it must be possible to navigate from the node of each interface to any other supported interface node. For this reason, every interface is required to have a common first method called `QueryInterface`. This method takes the name of an interface, and if the COM object supports it, it returns a reference to it otherwise it returns an error. Moreover, it requires that interfaces and their specifications are immutable, i.e. once published they cannot be changed. As a result, different versions of the same interface are considered as separate interfaces, allowing systems to concurrently support old and new versions easing system evolution. In order to be able to uniquely identify COM objects, every COM object is required to have an `IUnknown` interface and to provide the same interface pointer whenever queried for this interface (see Figure 2.3). As a result, the identity of this interface can be used as the COM object identity. The `IUnknown` interface includes also methods used to support reference counting, which is a form of garbage collection that requires the cooperation of all participating objects and is unable to deal with cyclic references. An interesting concept introduced by COM is the notion of categories. Categories are sets of interface identifiers and have their own identifiers (CATIDs), which are also GUIDs. A COM object can be a member of any number of categories and categories among themselves are totally unrelated. A category specifies not only which interfaces must at least be supported, but also which methods in these interfaces must at least be implemented.

Distributed COM (DCOM) [67], as the name implies, allows COM objects to be distributed over the network. This is achieved by using the client side proxies and server side stubs already present in COM to support inter-process communication between local object servers. In order to deal with platform differences in data representation, DCOM marshals data into a platform independent format (NDR). Furthermore, in order to form machine independent object references, it introduces object exporters, objects provided by DCOM that know how to bind the objects exported by a server. DCOM also provides higher-level mechanisms to speed up remote operations, provide security and detect remote machine failures.

At this point, we should note that as COM is a binary standard there is no need for an IDL. However, object proxies and stubs are needed both for local and remote object servers. In order to automatically generate these proxies and stubs Microsoft offers an
IDL compiler (MIDL). However, in order to use this compiler, interfaces need to be described in COM IDL. Note that the use of MIDL is not necessary since other tools like Visual C++, can produce the proxies and stubs directly. MIDL and these tools can also be used to create type libraries, which can be used at runtime to provide type information about interfaces and classes.

At runtime COM objects are supported by COM object servers that have defined structure and contain the COM classes they implement. These object servers execute within processes that can be separated into apartments. Apartments form separate synchronisation domains each with its own synchronisation regime. It is interesting to note that synchronisation actions are placed by the COM infrastructure at the boundaries of apartments. As part of their defined structure, COM object servers have a factory object for each class that the component contains. Besides the object servers’ factory objects, COM also defines a procedural library interface to request new object instances based on their CLSID. This interface utilises a system registry, which is similar to the CORBA implementation repository. For object initialisation the most direct way to initialise an object is to ask it to load its data from a structured storage, which is like a file system within a file. COM also defines a way to refer directly to a persistent object by name. Such names are called monikers and they are really objects in their own right referring to an object by specifying a logical access path. However, COM does not directly support persistent objects, as the object identity is not preserved.

Besides, the above-mentioned features, COM also provides a number of services. Important such services are:

- Uniform data transfer, which allows for the unified implementation of all sorts of data transfer mechanisms.
- Dispatch interfaces, which are similar to the dynamic invocation and dynamic stub interfaces in CORBA.
- Outgoing interfaces, which are either mandatory or optional interfaces that a COM object would use if it were connected to an object that provides this interface.

A related to COM but somewhat separate set of technologies is Microsoft’s compound document technologies. These technologies were first introduced in a more
primitive form in Visual Basic, where Visual Basic controls (VBXs) can be embedded into forms. A form binds the embedded controls together and allows the attachment of scripts that enable the controls to interact. This basic idea was subsequently extended and generalised with the introduction of Object Linking and Embedding (OLE) [112]. OLE distinguishes between document containers and document servers. A document server provides some content model and the capabilities to display and manipulate that content. A document container has no native content, but can accept parts provided by arbitrary document servers. Many document containers are also document servers. To accomplish its task OLE relies on the services provided by COM, and in fact is just a very large set of predefined COM interfaces. The idea of controls is also present in the form of OLE controls (OCXs), which despite their power compared to VBXs never became very popular. The main reason is that an OCX is required to implement a very large number of interfaces, including all OLE document servers' interfaces. So, in order to combine the benefits of COM supported controls, like OCXs, with the simplicity of developing VBXs, Microsoft introduced a new specification of OCX, namely ActiveX controls [71], which are just COM objects supported by a special server. Besides the regular COM interfaces, ActiveX controls also have outgoing interfaces, since controls signal changes by emitting events. Almost all ActiveX controls also have properties that a user or application assembler can use to fine-tune looks and behaviour. Moreover, ActiveX controls have to be implemented by a self-registering server, which when started and asked to do so, can register its classes with the COM registry. This is particularly useful when the server's code has just been downloaded from the Internet. Finally, an ActiveX container is an OLE container with a few additional properties.

Similarly to the various CORBA services, Microsoft offers a number of enterprise services for COM. These were originally part of the Microsoft Transaction Server (MTS) and the Microsoft Message Queuing (MSMQ). Later these services were combined with technologies for load balancing and clustering into COM+. The central idea was to use declarative attributes to separate concerns about infrastructure requirements from the code of components and applications. These attributes can then be used by the COM platform to provide contextual composition, in other words to intercept activities and inject appropriate calls to the infrastructure. COM+ combines MTS and MSMQ
declarative attributes with several new ones at application, component, class, interface and method level. It also provides a new attribution model for the automatic mapping between procedural invocations and message queuing, similar to message-driven Enterprise JavaBeans.

The second phase: .NET

The second phase in Microsoft's component platform evolutionary process is .NET. .NET comprises of the Common Language Runtime (CLR), a large number of frameworks, packaged into assemblies, and a number of tools. CLR is an implementation of the common language infrastructure (CLI) specification [111] adding COM+ interoperation and Windows platform access services. More specifically, COM – CLR interoperation is achieved by providing automatically synthesised wrappers that either present CLR objects via COM interfaces or COM object via CLR interfaces. Moreover, CLR offers dynamic loading and unloading, garbage collection, context interception, metadata reflection, remoting, persistence and other runtime language independent services. Similarly to CORBA, CLI defines a language-neutral platform. Unlike CORBA, but similarly to Java's bytecode and JAR files, CLI also defines an intermediate language (Common Intermediate Language - CIL) and a deployment file format (assemblies). Note, however that CLR instead of CIL uses the Microsoft intermediate language (MSIL), which is a CIL compliant superset. The added instructions support the additional CLR features. On top of that, CLI also provides support for extensible metadata.

More specifically, CLI comprises of the specification of execution engine services, like loader, JIT compiler and garbage-collecting memory manager, the Common Type System (CTS) and the Common Language Specification (CLS). The idea is that CTS includes all the CLI supported type concepts, which have been taken from many different languages, while CLS is a strict subset of CTS defined in such a way that a number of languages can support fully its type concepts. CTS distinguishes two kinds of types, value types and reference types, which are further divided into interfaces, classes, arrays and delegates. Classes are split into marshal by value and marshal by reference, which are further divided into context agile and context bound ones. CTS has no primitive types, and types like integer or floating point are just predefined value types. CTS supports multiple interface and single class inheritance. It is interesting to note that
value types can also implement multiple interfaces. An interesting feature of CTS is the ability of classes that implement multiple interfaces to qualify method names with the name of the interface that introduces them. This allows within a single class the concurrent use of two methods with exactly the same signature, a very useful feature for supporting side-by-side use of different versions of the same interface. CTS also supports naming conventions for properties and indexers. Finally, it does not include throwable exceptions as part of method signatures.

C# is an object-oriented language designed to be an exemplary .NET language. As such it directly supports most of CLR features [65]. C# supports the following kinds of types: interfaces, classes (CTS reference classes), structs (CTS value classes), enums (CTS enumeration classes), and delegates (CTS reference classes). There are some built-in types, like int or decimal, but they are not really primitive types, just normal value classes. Unlike Java naming conventions, C# introduces syntax for properties, which are abstract fields that have a get method, a set method, or both. Properties can be defined on interfaces, classes and structs. In addition to the abstraction of fields by properties, C# also abstracts arrays by indexers. However, there can only be one indexer per interface, class, or struct. In C# classes and interface can declare event sources, which are multicast, i.e. any number of listeners can register for them. Note however that support for events is just syntactic sugar and not special types. Finally, C# supports CLI custom attributes, which are user-defined meta-attributes attached to interfaces, classes, structs, fields, methods, parameters, etc. A lot of these attributes are defined in the various .NET frameworks.

As we mentioned above, CLI introduces assemblies as a deployment file format. In fact, in .NET assemblies are the units of deployment, versioning and management, in other words they are the .NET components. More specifically, an assembly is a set of files in a directory hierarchy. One of these files must be the manifest, which is a table of contents of the assembly. Assembly files can be divided into module files containing code in CIL and resource files containing immutable data. Assemblies can also be tagged with a culture label. An assembly is either private to a single application or shared among multiple applications.
Shared assemblies have to be uniquely named by a strong name, which includes a publisher token, an assembly name, a version vector and a culture. The version vector consists of major and minor version number, and a build and patch number. The publisher token is a public key (in reality a hash of the public key), while the assembly itself is signed by using the private key. This allows the receiver of an assembly to check the signature using the public key, thus ensuring it has not been tampered with. The way public keys are generated makes it very unlikely that two keys will be the same, thus almost guaranteeing that assemblies are uniquely named. Compared to the use of GUIDs in COM, strong names assign a unique identifier to the assembly as a whole and all contained definitions names are interpreted with respect to this strong name, while in COM instead every defined item is uniquely identified by a GUID. As a result, in .NET packaging decisions have a lasting effect on naming.

Every assembly also has to list the strong names of other shared assemblies they require. The fact that strong names are structured, including version numbers, combined with the fact that CLR supports the side-by-side use of the same assembly in multiple versions, allows for interesting configuration policies. These policies along with the shared assemblies themselves are stored in the global assembly cache (GAC). This is a facility provided by Windows that is effectively a database of assemblies keyed by their strong names.

At runtime the CLR execution engine partitions a process into one or more application domains (AppDomains). AppDomains define the scope of loading and unloading, and enforce a lightweight isolation boundary for all included objects. As a result, communication between AppDomains requires marshalling. At the same time, logical threads are the units of execution and are mapped to physical threads whenever they enter an AppDomain. AppDomains are further divided into contexts. Contexts are characterised by properties, which are shared by all object within the context. In addition to the support of contexts, CLI also provides runtime reflection support. This allows full access to the type structure of loaded assemblies, including all attributes and custom attributes defined on these types, as well as support for the creation of instances of these types and invocation of methods on these types. Note that CLI reflection is controlled by permissions using the CLI security framework. Furthermore, CLI provides also support
for remoting. This combines the context and the reflection infrastructure with flexible support for proxies, channels and messages to provide building blocks for a wide variety of communication styles and patterns. Unlike DCOM and like Java RMI, CLI support lease-based handling of remote references. Unlike Java RMI, remote references are not scoped to an entire JVM, but to an AppDomain.

Finally, one of the interesting frameworks supported by .NET is ASP.NET, which provides support for web applications. The support is provided in the form of Web Forms that produce Dynamic HTML (DHTML) [113] and rely on a remote browser to display forms and interact with users. The Web Forms framework fully abstracts the capture of an event on the client and its transmission to the server. A similar abstraction is provided for form state maintenance and the forwarding of updates to clients. Under the hood, what happens is that page descriptions are compiled into page classes that are dynamically loaded and instantiated. In contrast to the older ASP model, ASP.NET instead of just extracting code from web pages, interpreting it, running it and putting the result back into the page; it turns the whole page into an object which manages the interaction and produces the required HTML. This is very similar to way Java handles servlets and JSP. ASP.NET also supports the componentisation of page descriptions, which the use of pagelets. Pagelets allow modular use of re-occurring definitions and construction in the context of multiple pages.

2.3.4. Comparing perspectives: similarities and differences

The first thing we should note with respect to the above component platforms is that good ideas introduced in each one of them, over time have made their way to the others. The most striking example of this is support for attribute-based contextual composition. It was initially introduced as part of MTS in COM+ and has made its way to the EJB and CCM containers. Another such example is the provision of specialised component models for application servers and web servers, which was introduced by Java and has made its way to the .NET platform. As a result, there are a number of common features across all platforms. The most notable such feature is that all three rely on some sort of object model, which incorporates mechanisms for late binding, supported by object references, encapsulation of implementation details with the use of interfaces,
and support for inclusion polymorphism, i.e. subtyping. Additional features that are common across platforms are support for introspection and reflection, as well as persistence and serialisation.

At the same time, there are also some important differences between them, the most notable of which are:

- COM is the only platform that defines a binary standard. In contrast, Java and CLR stay at the level of intermediate language standards.
- Java is the only platform in which EJB containers support container-managed persistence and relations.
- COM is particularly strong in its support of evolution and versioning.
- CORBA is the only platform that does not provide any solution to the global memory management problem in distributed object systems. In contrast, COM provided a limited solution in the form of reference counting, and Java provides full support for distributed garbage collection.

Finally, the three platforms also differ in the extent to which they are supported by industrial strength platform implementations, development environments, and the provision of supporting services.

2.4. Concluding remarks

Component software for a long time has been the holy grail of software engineering research. As a result, a lot of research effort has been directed towards it. Despite some notable progress in making it a reality in software development, the questions of what a software component is and how can components be correctly composed are still open to debate. Therefore, it is important to delve into this debate in order to clarify what our view of software components is.

Within this debate Szyperski’s definition (see Definition 2.1) seems to not only capture the most important aspects of software components, but also the general consensus. This definition requires that components are independent units of deployment. However, this is interpreted by some to imply that they have to be binary pieces of software. In reality, the exact nature of components is not that important. What is important is that a strict encapsulation boundary is enforced hiding their implementation...
details, and that they are targeted to the deployment phase of the software lifecycle. Furthermore, the most controversial issue within this debate is whether software components are stateful or stateless. Although, there is not conclusive evidence that either view is more appropriate, the fact that state of the art component platforms consider stateful components tips the balance in favour of the stateful components view.

Besides the arguments within this debate, there are two other points that we need to keep in mind. The first point is that standardisation is expected to play an important role in component software. Therefore any work in the area needs to allow room for any standardisation developments to be easily incorporated. The second point is about COTS components, a particular type of software components that are also commercial entities, which has received a lot attention in recent years. However, in this thesis we take the decision to not focus particularly on this kind of components. As a result most of the work in this area, which is focused on how such component can be smoothly integrated into the software development process, is not particularly relevant to us.

The progress that has been made in component-based software engineering is particularly demonstrated by the state-of-the-art component platforms. These platforms are already widely used in the development of component-based applications, thus supporting our claim in section 1.1.2 that component software nowadays is more of a reality than ever before.

In conclusion, the discussion in the rest of the thesis relies heavily on the definition of software components provided in Definition 2.1, and we assume that the component location and selection service is supported by a component platform as the ones presented in this chapter.
Chapter 3. Semantically enhanced component trading (SECT)

In this chapter, we continue our contextualisation of the component location and selection problem. However, we focus our attention to the area of networked and open distributed systems where the problem of service location and discovery has been well studied and a number of alternative approaches have been developed, like naming, directory and trading services. Section 3.1 provides a presentation of these approaches. It should be noted that these types of services are nowadays widely deployed, with some of them having been even standardised by industrial bodies and international organisations. Our main intention is to establish which of these approaches is the most appropriate as a solution to our problem, which our analysis identifies as service trading.

Having identified service trading as the basis for the component location and selection service, we then present an in depth analysis of it in section 3.2. This analysis aims to determine the challenges that component software raises for service trading. The conclusion of this analysis is that a new kind of trading is needed, namely semantically enhanced component trading (SECT). We conjecture that SECT is able to both bring service trading into the domain of component-based development and in doing so to improve the quality of the service trading process in terms of precision and recall.

Moreover, this analysis also identifies the explicit description of behaviour as the main requirement for SECT. The description of behaviour is an area that has been extensively studied in research areas, like software reuse, in particular in the organisation and search of component repositories, and object interoperability. A survey of these areas is carried out in section 3.3. This survey provides the context for the development of a component type model for SECT. This type model enables SECT to support the component location and selection service described in section 1.1.2.

3.1. Service discovery and location in networked and open distributed systems

In section 1.1.2 we identified a component location and selection service as a crucial component of our new enterprise information system architecture. In fact, it is this service that makes possible the dynamic reconfiguration of enterprise information
systems, a core requirement for the support of the virtual enterprise model. Having specified what are the core properties of software components, and presented an overview of the current state of component platforms in the previous chapter, in this section we shift our attention to the problem of component location and selection. Actually, the problem has two separate but closely related aspects. On one hand, there is the problem of finding what components are available, i.e. deployed in our system, and from these components selecting those that are appropriate for our purposes, the new system configuration. On the other hand, having selected the appropriate components, there is the problem of contacting and configuring these components in order to construct the new system configuration. These two aspects of the problem are very similar to the service discovery and location problems in networked and distributed systems.

More specifically, in the case of networked and distributed systems, service location was considered a problem that required generic solutions, when it was realised that ad hoc ways of establishing communication between different services of a systems with all the details “hardwired” in the implementation of these services, led to rigid and fragile monolithic systems that were too difficult to manage and maintain. The proposed solution was to abstract away from these details by providing an underlying infrastructure, which could handle the communication between these services in generic ways. This is the purpose of service location protocols. Provided that the location of the communicating services is given, these protocols handle the exchange of messages between these services.

The location protocol requirement for services to know the network location of their communication partners becomes a problem in its own right. For the same reasons, described above, having the network location information “hardwired” in the implementation of a service is not really an acceptable option in most cases. If this is the case, then services need to have ways of discovering this information. The proposed solution was to incorporate in the underlying middleware, mechanisms that would allow this discovery to take place, referred to as service discovery protocols.

Service discovery protocols require services to somehow identify their communicating partners. There are generally two ways of service identification, name based and property based ones, which gave rise to naming and trading middleware
services respectively [116]. The former rely on a naming scheme that guarantees uniqueness of service names, and provide a mapping between these names and the location information of the services they refer to. In this case, the discovery process is driven by the description of the name of the sought after service. We could see this process as being analogous to the way we discover people’s telephone numbers using the white pages. The latter instead of assigning names to services, define different types of services, each accompanied by a set of properties. These properties are usually in the form of name values pairs, and describe the characteristics of services of the particular service type. In this case, the discovery process is driven by the description of the required service type together with its desirable properties. We could see this process as being analogous to the way we discover professionals’ telephone numbers using the yellow pages.

In both name based and property based discovery protocols, there are a number of alternative ways in which the discovery process takes place [124]. In general, the protocols can be classified into server based and peer-to-peer ones. In the former there is a server, name server or trader, which mediates the discovery process. Requests for services are sent to this server, which is also responsible of maintaining the mapping between names, or service types and properties, and service location information. This category raises the additional problem of how does a service know who to contact. Again, there are two options, either this information is provided to the service, in which case we are talking about pre-configured discovery; or the name service or trader is considered as just another service, in which case we can use a peer-to-peer discovery protocol to find it. Another problem of server-based discovery is how does the server find out about the available services. There are two alternative models that can be followed: push and pull. In a push model the service providers are responsible for registering their services with the server. In a pull model the server is responsible for gathering information about available services from the service providers. Combinations of both push and pull models are also possible.

In peer-to-peer discovery there is no server. Instead, services communicate directly to each other their location information. This communication takes place using some form of message multicasting or broadcasting. This category can be further divided
into passive and active protocols depending on who initiates the communication. In passive protocols service providers are responsible for advertising their services. Their clients are just listening passively to these advertisements until they identify the location information of a required service. In active protocols the clients are actively seeking the required services. In this case the service providers are the passive listeners of the requests and they provide their location information to the issuers of requests they can satisfy. The problem of keeping an up to date list of available services exists also in this case. Push, pull and combined approaches can also be used for this purpose. Moreover, besides the basic protocols described above more complicated protocol, including request or advertisement store and forwarding can also be used.

3.1.1. Current state of service location and discovery protocols

So far, the discussion has been on the general approaches to the service location and discovery problems. Before we examine the appropriateness of the various approaches as a basis for our component location and selection service it is worth providing a brief overview of the specific protocols currently available. However, first we should point out that there are two distinct perspectives in service location and discovery protocols, which although they follow the same basic ideas presented above, have produced quite different solutions. These two perspectives are: the networked systems one, and the distributed systems one. The former aims to address the challenges posed by mobile devices with networking capabilities and ad hoc networks, networks that do not really on a fixed infrastructure, in terms of configuration management. The main challenge being to reduce the amount of human intervention that configuration requires or in other words to provide zero configuration networking [125]. The latter perspective aims to address the challenges posed by open distributed systems, systems where the set of available services is changing over time and the system needs to continuously accommodate these changes.

The most prominent of the service location and discovery protocols developed from the networked systems perspective are:

- SLP (Service Location Protocol) [117] developed by the IETF (Internet Engineering Task Force) SvrLoc working group (www.ietf.org).
- Salutation [118] developed by an open industry consortium consisting of a large number of companies producing office equipment (www.salutation.org).
- Universal Plug and Play (UPnP) [119] developed by an industry consortium led by Microsoft (www.upnp.org).
- Bluetooth Service Discovery Protocol (SDP) [120] developed by the Bluetooth Special Interest Group, an industry consortium including a number of telecommunication companies and mobile phone manufacturers as part of the Bluetooth networking technologies for short-range wireless transmission technology. (www.bluetooth.com).
- Rendezvous [121] developed by Apple (www.apple.com).

All of these protocols with the exception of SDP address both service discovery and location, i.e. access to services. In fact, all these protocols with the exception of SDP support also notification mechanisms for monitoring the state of services. Most of these protocols with the exception of Salutation rely on a particular network transport protocol, SDP on Bluetooth, while SLP, UPnP and Rendezvous on TCP/IP. Salutation uses Transport Managers to allow it to use different network transport protocols. All these protocols support both server-based and peer-to-peer discovery and they all use property-based identification of services with the exception of UPnP, using a flat property space. UPnP is interesting in that it uses XML documents to describe the service properties, thus allowing their structuring. At the same time, SLP and Rendezvous use a URL scheme to specify service access points [122]. In the case of server-based discovery protocols like SLP, UPnP, Rendezvous and Salutations may use the Dynamic Host Configuration Protocol (DHCP) [126, 127] and DNS (Domain Name Service) SRV [128] for locating the discovery servers in a local area networks and given DNS domains respectively.

On the other hand, the location and discovery protocols developed from an open distributed systems perspective are closely associated with particular middleware platforms that enable some level of distribution transparency. In recent years, the most prominent of these platforms are distributed object platforms like CORBA, Java with its RMI, DCOM and .NET (see section 2.3 for more information). However, closely related and with respect to service location and discovery probably more interesting are also platforms like ANSA [42] and ODP (Open Distributed Processing) [11]. The reason is
because the former introduced concepts like service trading [41], the predominant approach for service discovery in distributed systems, while the latter has standardised specifications of such services (e.g. [43]) that more recent approaches follow.

More specifically, in these protocols location and discovery are usually kept separate. The location problem is addressed by the use of some kind of service references, which encode the location information of the service (e.g. CORBA object references, or Java RMI references). These references are created by the middleware platform when the services are initialised. In this approach it is also quite common to introduce a level of indirection, which enables looser coupling of the communicating services, along the lines of the broker pattern [123] (e.g. the CORBA object request broker, or the Java RMI registry). Service providers register their services with the broker, which creates the service references. When clients want to communicate with a service they do so through the broker using the service reference. The broker is able to translate the provided reference into network locations and to route the communication to the referenced service.

For the discovery problem, these protocols tend to favour the server-based approach using either a name server or a service trader. There are a number of reasons for this. First, these protocols are usually targeting enterprise information systems, which contrary to ad-hoc networks have fixed networking infrastructure they can rely on, and clear administrative authorities that can assign and maintain central services. These characteristics mean that these systems are more amenable to server-based approaches than the mobile and ad-hoc networking ones. Second, in enterprise information systems, there are both a larger number of services and a lot more variety of service offers, leading to more demanding discovery process that has to be supported by more powerful devices. As it is not realistic to expect all devices in a distributed system to be that powerful, server-based approaches seem more appropriate.

Jini: an interesting perspective to service location and discovery

One particular service location and discovery protocol that deserves separate mention is Jini [129]. The reason is that Jini has received a lot of publicity, and is probably the only protocol that although it was developed from a networked systems perspective, enjoyed a lot more success in the distributed systems arena. Jini in general
defines both an architecture and a programming model for networked systems. The architecture is built upon a number of protocols, which allow service providers to enter a network of services in order to make their services available [130]. These protocols are: discovery, join and lookup. The idea is that in order for a service provider to join a network of services, it needs to find a lookup service to register its services. After the registration of the services, clients can find and use them by contacting the lookup service. More specifically, the discovery protocol describes how services can find lookup services to either register or query. The join protocol describes how service providers register their services, while the lookup protocol describes how client can query a lookup service to discover the service they need. So, it is important to point out that the lookup protocol is in fact the service discovery protocol of Jini. The protocol is server-based and uses property-based identification of services, where the properties are described using property objects.

The most interesting feature of Jini is that instead of just registering and passing around location information, it uses service objects. The service objects encapsulate the communication protocol that service clients and providers need to follow. These objects are created by the service provider, who passes them, as part of the join protocol, to the lookup service; which in turn passes them to the service clients, as the result of the lookup protocol. The service client instantiates these objects and uses them as proxies of the service provider [114]. The major benefit of this approach is that the whole interaction between service clients and providers becomes communication protocol independent [131]. More importantly from a networking point of view, in order for services to interact with newly introduced devices, they no longer need to have a device driver available. The service object is in fact the device driver.

However, the whole approach was strongly criticised because it seemed to require that all devices are Java enabled, i.e. run versions of the Java Virtual Machine (JVM), which for a lot of devices is unrealistic. In reality, this was a misconception. Jini defines a surrogate architecture, which removes the requirement for all devices to run a JVM. The idea is that devices with limited capabilities will use a surrogate host as an intermediary, when participating in the network of services. In fact, Waldo, one of the Jini architects, goes even further and claims that the only thing that is needed is for devices to participate
to the network of services, is the ability to register a Java object, their service object, with the lookup service, while service clients need to be able to download service object code, load it and call it [131]. In other words, the JVM as such is not really necessary for the Jini protocols, instead certain features like creation of service and property objects and their dynamic loading and activation is all that is required.

Although this kind of criticism was shown to be mostly unsubstantiated, it still managed to hamper the acceptability of Jini for networked systems. At the same time, its core features made it appealing for the development of more flexible and dynamic distributed systems. Besides the use of service objects, another particularly appealing feature of Jini is leasing. Leasing allows service providers to grant access to their service for a particular period of time. If a service client needs to continue using the service for longer than the lease period, then it needs to explicitly renew the lease. In this way, service providers are always aware of when their services are needed, which is particularly useful for the dynamic reconfiguration of the systems with the removal of old services and the introduction of new ones.

In conclusion, the main contribution of Jini has been twofold. First, Jini introduced notions like service discovery and leasing in the already popular Java platform. As a result, these notions are now part of the mainstream of distributed systems development. Second, it exploited particular features of the Java platform to provide an innovative solution to the service location and discovery problem, namely service objects. In fact, the use of service objects for service discovery makes service location a non-problem, because it removes the need for service clients to directly communicate with the service provider. Clients only need to communicate with a local proxy of the service, the service object, and don't really care where the service is located. However, what is really important is that all this is done without the need to specify a communication protocol like for example through an Interface Definition Language, as is the case in platforms like ANSA, ODP and CORBA.
3.1.2. Trading as the basis for a component location and selection service

In the previous sections, we first examined the service location and discovery problems, and then presented a number of alternative solutions to these problems and how these solutions have been incorporated into current service location and discovery protocols. In this section we examine the appropriateness of these solutions as a basis for a component location and selection service. The discussion is almost exclusively focused on solutions to the discovery problem, because we believe that the location problem is mostly taken care of by the underlying component platforms.

More specifically, as we have seen in section 2.3, current component platforms build upon and extend pre-existing distributed object platforms. Furthermore, as we discussed in section 3.1.2, these distributed object platforms incorporate solutions to the location problem involving notions of distributed object references. In fact as we have seen, they usually go further and use object brokers and automatically constructed local service proxies to completely remove any concerns regarding the location of communicating services. Therefore, this kind of solutions to the location problem is already available in current component platforms. However, from a component software point of view, there is a requirement for more dynamic configuration and reconfiguration of components compared to what traditionally has been the case for services supported by distributed object platforms. This is one area where the extensions introduced by current component platforms play an important role. First, there is the introduction of component containers with their support for contextual composition. As we have seen, containers provide an attribute-based model for the configuration and in a lot of cases the dynamic reconfiguration of the contained components. Second, there is the use of home interfaces (see particularly OMG's CCM and Sun's EJBs). These interfaces enable the activation and the initial configuration of the types of components they support. For these reasons, we focus the rest of the discussion on the discovery problem. Provided that our enterprise information system architecture (see section 1.1.2) is built on top of such a component platform, then the role of the component location and selection service becomes to find which components are appropriate for our needs, and to provide us with some sort of reference that would enables us to contact their home interface.
We now have a clear context, within which to examine the various solutions to the discovery problem. The first issue we need to consider is the choice between name-based and property-based approaches. It is easy to see that using names has clear advantages compared to using directly some kind of reference to identify components. First, names can be independent from the location of the component and second they are easier for people to remember. However, as the number of available components increases, managing and remembering a large number of names becomes problematic. As a result, the discovery process begins to suffer. In order to improve the scalability of name-based discovery, a common approach has been to divide up the name space by introducing some kind of structure to it. Consequently, the names used are also structured, so as to reflect the underlying structure of the name space. This idea led to the introduction of directory services [132, 133, 134], which impose a hierarchical structure to the name space and follow a hierarchical naming scheme. In such a naming scheme, names comprise of a number of elements, corresponding to a path of the hierarchy.

In order to demonstrate how directories improve the scalability of the discovery process let us consider again the case of using the white pages to find somebody's telephone number. If there was no structure at all to the white pages, then finding the telephone number we want is almost impossible. If on the other hand the names are organised alphabetically, then the whole process becomes a lot more manageable, but it may still be quite time consuming. For example, if we have to search through a single phonebook for the whole country, chances are that there are going to be a lot of people with similar names, thus requiring some time to find the one we are looking for. However, if on top of the alphabetical ordering of names we also divide the phonebook into geographic areas then the procedure becomes even more efficient. In general, introducing additional structure allows us to quickly cut out larger parts of the search space and as a result improves the efficiency of the search. The best argument for the scalability of directory services is the use of the Domain Name Service [132] for identifying hosts on the Internet.

Although with the use of directory services the discovery process scales quite well, the reliance on names still presents some problems. First, despite the fact that names are organised into a hierarchy, there is still likely to be a large number of them, which
means that it is still difficult to remember them all. Second, although we can use similar
names to reflect similarities between the identified components, using such an approach
does not reveal anything about the extent of similarity or difference. In an attempt to
address this particular problem and also to provide more information than just location in
service discovery, directory services started to support the association of sets of attributes
to the services (e.g. [133, 134]). However, the most serious problem in name-based
discovery is that clients need to know in advance the names of all components they may
use. This kind of requirement restricts seriously the extensibility of systems, since the
introduction of a new component requires that all its potential clients be updated. For all
these reasons, name-based approaches to component discovery do not seem appropriate.

As we mentioned above, in service discovery property-based approaches tend to
rely on some notion of service type. Instead of identifying services by name they identify
them by their type and the particular properties they have. The use of service types is
what makes them fundamentally different to directory services including service
properties. This is because the service type carries information regarding the capabilities
of the service. The whole discovery process shifts from having to remember the names of
services or components, to just having to express your needs in a standardised fashion.
Using property-based discovery you find services and components that can perform
particular functions instead of ones having a particular name. Since the names of the
components become irrelevant, then no matter how many they are, developers do not
have to remember them. More importantly, since the whole process now relies on client
needs, there is no need to update clients whenever new components are introduced. If
they provide the required functionality, the new components will be discovered and used.
Furthermore, similarities and differences between particular components can be clearly
expressed as relationships between their corresponding types. For all these reason, we
believe that property-based discovery is the most appropriate for the component location
and selection service.

Having selected property-based discovery as the basis for the component location
and selection service, the next choice is between server-based and peer-to-peer
approaches. As we mentioned in the discussion of the different perspectives in location
and discovery protocols (see section 3.1.1), server-based approaches tend to be the norm
in enterprise information systems. The same reasons behind this choice could also justify our preference for a server-based approach. In addition to these reasons, in the case of property-based discovery in the context of software components of various levels of granularity, we would expect a larger variety of types and relationships between these types to be both available and utilised. As a result, the searching process would have to be more complicated than in the case of services. This further reinforces the need for a powerful server, able to handle the additional complexity. In the context of service discovery, traders have been designed for exactly this purpose. Therefore, we believe that trading is the most appropriate basis for the component location and selection service.

3.2. A closer examination of service trading

The ANSA model [42] for open distributed processing introduced the notion of service trading with the specification of its trading facility [41]. It defined trading as:

Trading is the activity of choosing services, such that they match some service requirements. The choice is based on the comparison of the specification of a service required (provided by a prospective consumer) and the service specification supplied by service providers or their agents.

Definition 3.1. ANSA Trading.

The entity responsible for carrying out this activity is called a trader. The idea of trading was then adopted and standardised in the trading function [43], part of the basic reference model for open distributed processing (RM-ODP) [11]. At the same, the OMG took on trading and introduced it as one of the object services of its Object Management Architecture (OMA) [24, 56]. In their basis all three specifications are quite similar. However, the RM-ODP one is certainly the most comprehensive. Furthermore, at the time of its standardisation it attracted a lot of scientific interest and a number of implementations were produced, e.g. [135, 136, 137, 138, 139]. Despite this interest, it was the OMG trading service, which made service trading part of the mainstream as it was one of the most common object services to be included in commercial CORBA implementations (e.g. [140, 141, 142]).

More specifically, the trading process involves the following steps (see Figure 3.1). First, service providers export their services. This involves registering offers of their
services with the trader. Service providers are commonly referred to as exporters. Then, service consumers import particular service offers. This involves requesting from the trader the required services. Upon receipt of a request, the trader searches its collection of available service offers in order to find ones matching the required services. If any such offers are found they are returned to the requester, which is commonly referred to as importer. In fact, the trader may select particular offers according to criteria provided by the importer and only return them. From this point onwards the importer can directly communicate with the exporter without the participation of the trader.

![Trading Architecture](image)

Figure 3.1. Trading Architecture.

In order for exporters to register their offers with the trader, they need to provide a reference that clients will use to invoke the service operations. This reference in the case of ANSA is an interface reference, while in the case ODP and CORBA is an object reference. In addition to this, they also need to provide the type of the exported service. In the case of ANSA, this means to provide an interface type, which specifies the names of operations to which the service responds along with their parameters and result types, i.e. the operation signatures. Besides, the interface type ANSA also requires a set of service properties, which are distinguishing characteristics of a service not expressed by its interface type.
On the other hand, ODP and CORBA introduce the notion of service types, which are defined by an interface type and a set of service properties types. Similarly to ANSA, the interface type specifies the names and signatures of the operations the service supports. Meanwhile, the service property types determine the set of allowed properties. As a result, when exporting a service, the exporter needs to specify its service type and to provide values for the service properties, in the form of name-value pairs. In the case of CORBA service properties may also be declared as either mandatory or optional, in which case an exporter needs to provide values for at least all compulsory properties, or even as read-only. Note that because ODP and CORBA specify service properties as instances of particular property types, the trader can check if the values provided by an exporter are acceptable. In the case of ANSA this is not possible, which could also be the case for some service properties in CORBA, since it allows exporters to provide additional properties beyond those specified in the service type. In all cases, property values can either be static or dynamic. In the former case, the trader just stores the values, while in the latter case it obtains them on demand.

Moreover, all specifications include a notion of trading context. Trading contexts are used to organise and divide the trading space. In general, a trading context is just a set of service offers. ANSA requires that these offers are under the authority of a particular administration, while ODP and CORBA does not impose any such constraint. Accordingly, exporters need to also determine the trading context of their offers. Placing a service offer within a particular context restricts its visibility during the trading process, which only considers particular contexts. ODP also allows properties to be associated with trading contexts. These properties may describe some aspects of the characteristics of the set of offers itself; some common properties of the service offers; or even some common properties of the corresponding services. These properties can be used as the basis for the placement or searching of service offers.

Besides the above information, ODP also requires exporters to identify themselves by providing a client identifier. Moreover, it allows them to specify offer properties, assertions for a particular offer, quite often placing restrictions on the use of the particular offer. Finally, it also allows exporters to provide an optional interface identifier, which enables them to exercise additional control to the trading process by
providing additional guidance for the selection of an exported service, or values for
dynamic service properties including availability of service.

In order for importers to request a particular service, they need to specify what are
their requirements. In the case of ANSA this means that they have to provide a type
description that tells the trader what operations the application is going to expect to be
able to invoke on the obtained service. Instead, in the case of ODP and CORBA service
type identifiers are used for this purpose. ODP allows also the exporter to provide just the
interface type of the required service. In addition to the service or interface type, all
approaches require that the importer also specify a set of constraints on the acceptable
values of service properties, referred to in ODP as a matching constraint. In fact CORBA
has specified a particular constraint language for this purpose. In the case ODP, similar
constraints can also be placed on offer properties. Moreover, all approaches require the
importer to also specify the scope of the search for matching offers, which is limited to a
set of contexts.

ODP and CORBA allow also the importer to specify optional selection criteria or
preferences. These allow the trader to order the matching offers before returning them to
the importer. In the case of CORBA preference rules are predefined and the importer can
only choose one of them, e.g. random, fist, order by property value, etc. The default is
first, i.e. return matches in the order they were found. Furthermore, in the case of ODP
the importer is required to provide its identity and may also provide a search method that
determines how the search for matching offers should be carried out.

According to Definition 3.1 trading is basically the process of matching service
offers to service requests. For an offer to match a request it must have a type that
conforms to the requested one. In the case of ANSA, this refers to interface type
conformance with the additional requirement that the properties of the service also match
the constraints specified. ANSA interfaces provide a number of distinctly named
operations each with one or more possible terminations, while operations carry zero or
more arguments, and terminations carry zero or more results. Interface types describe the
names and parameters of each of the operations and the results of each of the
terminations. Interface conformance is based on the “no surprise rule”, which dictates
that in an interaction between a service and its client both sides must not be surprised by
unexpected input, output or termination conditions. Consequently, a conformant interface should provide at least all the operations with the specific arguments and terminations with the specific results that the client declared to be expecting in its import request. These conditions are captured by the four recursive rules of conforms that Deschrevel provides in [41]. The four rules are:

- **Interface conformance.** The type of interface \( I_1 \) conforms to that of \( I_2 \) if and only if for each operation \( O_2 \) in \( I_2 \) there exists an operation \( O_1 \) in \( I_1 \) with the same name whose operation type conforms to \( O_2 \).

- **Operation conformance.** The type of operation \( O_1 \) conforms to that of \( O_2 \) if and only if the parameter list type of the arguments of \( O_2 \) conforms to the parameter list type of the arguments of \( O_1 \), and the response type of \( O_1 \) conforms to the response type of \( O_2 \).

- **Response conformance.** The type of response \( R_1 \) conforms to that of \( R_2 \) if and only if for each termination \( T_1 \) in \( R_1 \) there exists a termination \( T_2 \) of the same name in \( R_2 \), and the parameter list type of \( T_1 \) conforms to that of \( T_2 \).

- **Parameter list conformance.** The type of parameter list \( L_1 \) conforms to that of \( L_2 \) if and only if \( L_1 \) and \( L_2 \) have the same length and the type of each interface in \( L_1 \) conforms to that of the corresponding interface in \( L_2 \).

In the case of ODP and CORBA, type conformance between offers and requests refers to service type conformance. However, as service types are specified by an interface type and a set of property types, service type conformance is defined as conformance of the corresponding interface and property types. Therefore, offer and request conformance in ODP and CORBA also relies on rules similar to the above four. Note that we are talking about similar rules because ANSA, ODP and CORBA consider different kinds of interfaces. More specifically, ODP distinguishes three kinds of interfaces [11]: operational interfaces, which consist of a set of operations and are similar to the ANSA interfaces; stream interfaces, which consist of a set data flows, i.e. continuous flows of data; and signal interfaces, which consist of a set of signals, i.e. atomic messages. Whereas CORBA interfaces consist of sets of methods and are described using CORBA IDL (see the OMG perspective in section 2.3).
It is important to note that in either case type conformance defines also a subtyping relationship between service and interface types. Subtyping is usually defined with reference to the principle of substitutability expressed by Wegner and Zdonik in [9] as: “An instance of a subtype can always be used in any context in which an instance of a supertype is expected”. It should be clear that the principle of substitutability is related to the “no surprise” rule on which the definition of interface conformance is based upon. In fact, if we specify types in terms of interface definitions, as is the case for service types, then it should be easy to see that the substitutability principle and the “no surprise rule” are equivalent. This equivalence is clearly demonstrated in the CORBA specification where the “no surprise” rule is expressed in terms of substitutability [56]. As a consequence, the service and interface conformance relationship is in fact a service and interface subtyping relationship respectively. As a result, in ODP and CORBA traders, service types are related in a hierarchy that reflects interface type inheritance and property type aggregation. Then during the trading process, this hierarchy provides the basis for deciding if a service of one type may be matched to a request of another service type.

In addition to the conformance of offer and request types and the satisfaction of the property constraints, in the case of ODP and CORBA, the trading process is also constrained by import and export policies. Each importer can specify an import policy that restricts the set of service offers considered by the trader, while each exporter can specify an export policy that restricts the set of importers to which a trader may convey a service offer.

From the above discussion, it becomes apparent that service and interface types and their relationships are at the heart of the trading process. Moreover, exporters and importers refer to these types by name in order to specify the type of their offers and requests respectively. Consequently, we can view traders as executing two distinct functions: type management, and offer management. However, as the number of available types grows, managing the types, their relationships and their descriptions becomes a challenging task. In open distributed systems type managers have developed in response to this challenge [39, 4]. As a result, trader architectures have been proposed that utilise such type managers to accomplish their tasks [1]. In general, type managers
provide two functions [143]: (a) type description, which provides operations to describe and compare types, and (b) type management, which provides operations to record type names and their relationships. In order to support these functions the type manager requires the definition of a type model that specifies: (i) the types it contains, (ii) how each type is described, (iii) a scheme for naming the types, (iv) the relationships that may hold between types, and (v) axioms and inference rules for deducing properties of the types. Similarly to traders that maintain offer repositories to support them in managing service offers, type managers maintain type repositories. In fact, such a type repository is in the process of being standardised as part of ODP [144]. Whereas, the CORBA Interface Repository also serves a similar purpose. Finally, besides their support for the trading process, type managers in open distributed systems can also be used to carry out type checking of remote references or to support introspection of service and interface types.

In order to deal with the problems that centralized and universal administration can cause to large-scale distributed systems, all trading specification support the federation of traders administered by different authorities. According to Deschrevel [41], there are four principles that characterize the federation of administrations:

i. An administration must not be forced to perform an activity for another administration.

ii. An administration must have the freedom of association with respect to the federation.

iii. Each administration determines which resources it wishes to share with other administrations.

iv. Each administration determines how it will view and combine the resources that are made available by other administrations.

For example, CORBA defines links to represent the paths for the propagation of queries from a source trader to a target trader. Importers may specify as part of their request a limit on the number of links that may be followed, when searching for matching offers. At the same time, links themselves have an associated policy, which overrides the importers link follow policy and places an upper limit on the number of links that may be followed. Furthermore, the CORBA specification also includes the optional functionality
of proxy offers, which are a cross between service offers and a form of restricted links. In the sense that although they are matched in the same way as normal offers, when they are matched, the query request (modified) is forwarded to the trader's interface associated with the proxy offer. The federation architecture defined by CORBA is only one example of how multiple traders can be utilised. In practice a number of alternative architectures are possible. Vasudevan in [145] identifies the following alternatives: non-communicative federation, where multiple independent traders are deployed for load balancing purposes; repository-based federation, where multiple traders share the same service offer repository; direct federation, for example the link architecture defined by CORBA; type manager based federation, where traders from different domains are managed by type managers that can translate between the different domains in some cases using interceptors [123]; and heterogeneous federation, where proxies [114] are utilised to federate traders from heterogeneous middleware platforms.

In conclusion, in service trading exporter and importers rely on explicit or implicit notions of service types in order to describe their offers and requests respectively. Service types are usually specified by an interface type and a set of property types, where the interface type describes the supported operations in terms of their names and signatures, and the property types express characteristics of services not captured by their interface in the form of name-value pairs. However, reference to interfaces and interface types requires some care, as there a number of different kinds of interfaces. The trader organises service types into subtype hierarchies according to some notion of service or interface type conformance. Subsequently, the matching process uses these hierarchies to find offers of service types that conform to the requested service type. As a result, type management is a central function of any trader, which relies on the specification of an underlying type model for the managed types. Finally, traders can be combined into federations according to a number of alternative architectures in order to improve the scalability of the trading process.

### 3.2.1. Components and trading

Having studied, in the previous section, service trading in detail, in this section we continue its examination in order to determine how it can be adapted to support a
component location and selection service. This examination is carried with respect to our adopted definition of software components (see Definition 2.1), and identifies the implications to trading for each of the elements of the definition.

More specifically, according to the definition "a software component is the unit of composition ...", as a result service trading needs to change into component trading. This requires that a notion of component types is introduced and the trading process will have to be carried out in terms of these component types. In general, a type categorises entities (services, objects, components) according to their usage and behaviour [146], or in other words a type is a partial specification of behaviour in some domain of discourse [147]. As a result, the notion of component type should in some sense specify component behaviour. At the same time, as we saw in service trading the service types are specified in terms of interfaces, which encapsulate their implementation details. This kind of encapsulation can also be useful in the case of components, and is in fact the reason why interfaces are explicitly mentioned in their definition. Consequently, it makes sense to specify component types in terms of interfaces. However, we should note that the kind of interfaces used in service trading is not really appropriate, since it does not capture service behaviour. Kiniry makes exactly this point when he says that the "... behaviour of the methods is clear only by explanation or documentation, and therefore susceptible to misinterpretation and misunderstanding" [151].

Moreover, the definition requires that these interfaces be "... contractually specified ...". On one hand, any interface can be considered as describing a contract that clients of a service or component should follow, in the sense that it at least specifies the messages that the service or component will respond to and what kind of response it may provide. On the other hand, Digre draws attention to the point that although this may be sufficient in some cases, e.g. in infrastructures like CORBA or Java RMI, "an interface definition ... is insufficient for frameworks that support semantic concepts beyond those provided by a messaging infrastructure" [148]. Therefore, in the context of component-based development, more comprehensive notions of contract are required like the ones Meyer introduced as part of contract-based programming [149]. These contracts require the specification of both the obligations and the benefits for each collaborating party in much the same sense as a legal contract, and are described in terms of preconditions,
postconditions and invariants. There is now agreement that this kind of contracts is the absolute minimum for component-based development [95, 97], while at the same it is argued that they should be extended to cover other aspects of component behaviour, particularly extra-functional ones [84].

The next part of the definition requires that a component should have "... explicit context dependencies only". This requirement has a number of consequences. First, it means that besides what they provide, components also need to specify what they require. In service trading this has traditionally not been the case. It is interesting also to note, that even state of the art component platforms like EJBs only include limited provisions for describing component requirements. Component requirements can be largely expressed in a similar manner to its provision, i.e. in terms of a required interface. However, the component definition also requires that the required interface is complete, in the sense that it describes all requirements. As a result, information about which events are produced and consumed, and the states the component exposes, which have been traditional ignored in service trading need to also be specified. The most important, though, implication is about the semantics, i.e. the meaning, of required interfaces. As we mentioned in section 3.2, in service trading interfaces include both the names and the signatures of the supported operations, while the definitions of operation and response conformance require exact names. However, as Wills points out, "the only reason we accept lists of operations as [service] specifications is that the names usually suggest the expected behaviour" [150]. In other words, names are used to implicitly convey the operation semantics. Although this may work in domains with well-established and precisely defined vocabulary, e.g. mathematics were the name of each function has a clear and precise meaning, in the general case it can be the source of mistakes and misinterpretations. Even more so, in cases where systems may comprise of independently developed components from a number of different non-standardised application domains. At the same time, the implicit description of semantics is also in contrast to the requirement for explicit context dependencies.

The problem of implicit semantics is not only limited to required interfaces, but also applies to provided ones too. As the number of type names grows and developers attempt to capture fine-grained differences between types by using quite similar but not
exactly the same type names, the exact meaning of the names becomes very difficult to discern [44]. The implicit semantics start to get lost and the effectiveness of the trading process is seriously affected. Traders will most likely fail on one hand to capture any relationship between similar service type names, synonyms, and on the other to detect inconsistencies in the use of specific names, homonyms. As a result, we need a notion of component type that explicitly describes the behaviour of the components. This is exactly the argument that Wills makes when he argues for rigorous methods that support "...a concept of type that includes descriptions of behaviour as well as the bare signature" [152]. The introduction of this kind of types is one of the main features of the Catalysis approach to component-based development [21] that he and D'Souza developed, and which "...formalizes the signature-based notion of an interface" [153]. Gennari et al. were some of the first to recognise the need for this kind of types in the context of component-based development [154]. They identified "the inclusion of semantics ...: information about the meaning of a component, information about what the component will accomplish, and information about the method's inputs and outputs", as the primary addition in order for CORBA to provide a solid basis for component-based development. The final requirement for components to be "... subject to composition by third parties", parties that have no knowledge of both the component's internals and the internals of the component's clients, further reinforces all the above requirements. In particular, it means that the component type specification should be described with adequate precision to eliminate the chances of any misinterpretations and misunderstandings.

In conclusion, the main problem of service trading with respect to component software is its restricted notion of types, which lacks explicit descriptions of behaviour, i.e. type descriptions are syntactic in nature. As a consequence the trading process is based on syntactic notions of conformance and a fairly restricted interpretation of the "no surprise" rule, which leads to an overall trading approach that is syntactic in nature. In contrast, the component location and selection service requires trading based on a notion of types that includes explicit descriptions of behaviour, i.e. type descriptions that are semantic in nature. Such types will enable the trading process to exploit also semantic notions of conformance, where "[c]onformance is about what you can do with a [component] and how it responds" [152]. This shift of focus from syntax to semantics
combined with the addition of component types, introduces a new kind of trading, namely **Semantically Enhanced Component Trading (SECT)**.

### 3.3. The solution in context

In the previous section we analysed service trading from a component software perspective and we identified its syntactic nature as its main problem in supporting the component location and selection service. We also introduced a new kind of trading, SECT as the solution to the problem. SECT enhances service trading with the inclusion of component types that explicitly describe not only the syntactic but also the semantic features of components, thus allowing the introduction of semantic notions of type conformance into the trading process. As we saw in section 3.2, traders rely on type managers for the management of their types and type relationships including type conformance. Type managers in turn rely on a particular type model to carry out their operation. Consequently, the realisation of SECT requires the definition of an appropriate component type model that incorporates semantic notions of both types and type relationships. A type manager utilising this component type model will be at the heart of a semantically enhanced component trader. In order to develop such a component type model we need to answer the following questions:

- How are component types in general, and component semantics in particular described?
- How are semantic component type relationships in general, and component conformance in particular defined?

These questions have been the focus of a lot of research in the area of software reuse. In software reuse, as the name implies, components, reusable software elements, are developed and subsequently made available for future use through component repositories. Such a process relies on the availability of techniques to describe the behaviour of components and to select components from a repository that exhibit particular behaviour, namely component retrieval. A variety of such techniques are currently available. We examine these techniques in section 3.3.1. At the same time, the above questions have also received a lot of attention in the area of object interoperability. The aim of this research has been to identify and describe the object characteristics
necessary for the seamless interoperation between objects. Much of this work is aimed at the in depth exploration of the “no surprise” rule and the extent of its implications. We examine the results of this work in section 3.3.2. In general, this section sets the frame within which the component type model for SECT will be developed.

### 3.3.1. Component selection in software reuse

As we mentioned above, there has been significant research in encoding component behaviour in the component retrieval phase of software reuse [157]. The proposed solutions fall into three categories [158]: text-based, lexical descriptor-based and specification-based. Text based solutions use the textual representation of a component as an implicit description of its behaviour, while employing arbitrarily complex string matching expressions to retrieve required components. Although they have low maintenance costs and are easy to introduce, a textual representation does not guarantee sufficient information for the classification and in fact could be misleading.

Lexical descriptor-based solutions use key phrases, which are constructed from a predefined vocabulary provided by subject experts, to describe what a component is. This technique can be extended to describe a number of different aspects of the component, leading to a technique commonly called multi-faceted classification [159]. The use of key phrases, which are assigned by subject experts, makes the method sounder and more complete. But, the construction of the predefined vocabulary is not a trivial task and there is also an ambiguity on the type of semantics (computational or application ones) that the vocabulary should describe. Moreover, there have been also proposals that try to enhance lexical descriptor-based approaches by formalising the relationships between the used keywords. An example of such an approach Concept-based Component Retrieval [178], which is based on formal concept analysis that organises concepts in a lattice of super-/sub-concepts.

Specification-based solutions use a specification language, whose semantics define the classification and retrieval scheme. In fact, “specification-based retrieval comes closest to achieving full equivalence between what a component is and does and how it is encoded [described]” [158]. There are a number of specification methodologies, which deploy various degrees of formality, from informal ones [160, 161, 230, 21] to
formal ones [162, 163]. Specification-based approaches are in general more powerful than both text and lexical descriptor based ones, mainly because of the wide range of formality they offer, making them preferable as a basis for the SECT type model.

The issue of formality

Having selected specification-based approaches to component classification and retrieval as the most appropriate for the SECT type model, we need to examine further the issue of formality. When selecting the appropriate level of formality, there is a trade-off between precision and usability. Formality provides precise, complete and consistent descriptions of components, besides "the only way to eliminate ambiguity is to be formal" [164]. As we saw in section 3.2.1, the requirement of Definition 2.1 for software components to be "... subject to composition by third parties", requires precise description of their behaviour. Consequently, a formal approach would seem more appropriate. However, the complexity of formal specification languages makes them difficult to use limiting their popularity [165]. Finney has carried out a study that further supports this argument [179]. It showed that people with a typical training in formal methods have difficulty in understanding fairly simple formal specifications.

In order to address the usability problem, two particular approaches have been developed: lightweight formal methods [181] and the formalisation of informal but popular specification languages [184]. The former approach usually trades off completeness and language functionality for efficiency. Moreover, they usually replace part of the mathematical notation with diagrams, which are in general easier to construct and understand. A typical example of such a specification language is Alloy, which comes with the Alloy Constraint Analyser (Alcoa) [182] that allows the detection of specification defects early in the development lifecycle. For the alternative approach the most promising candidate is the Unified Modelling Language (UML) [161]. Since its adoption by the OMG, UML has become a de facto standard for object-oriented modelling, while it is already in the process of becoming an ISO standard too. UML provides a graphical notation for system modelling complimented by the formal Object Constraint Language (OCL) [32]. Furthermore, significant effort is put to provide precise semantics in UML under the name of pUML [184]. In general, both approaches can be seen as opposite sides of the same effort to convergence UML and formal methods by
either UML adopting formal methods ideas that provide implementation abstraction and verification capabilities or formal methods becoming more UML-like. This view is further reinforced when we consider Jackson’s claim that “Alloy is close enough to UML to make transcription of an object model diagram into Alloy a trivial task...” [183].

Another criticism of formal specification techniques is that precise specification is not cost effective [180]. They require significant effort for a very low pay back, since most of the produced software is not reused. However, this argument makes little sense in the case of software components, where most of their benefits come from their extensive reuse. In the case of software components precision and detail pays off [185], since higher quality components are more likely to be used and mistakes may be very costly [186]. Moreover quite often, formal techniques are perceived as costly because the development of the specification of the system and its code are usually separate activities requiring different languages. With the move towards more UML like specification languages this is no longer the case, as UML tools are often able to produce code from the diagrams and to automatically maintain the consistency between code and specification. At the same time, another approach, which gained a lot of popularity in recent years, is the merging of programming and specification languages supported by a variety of appropriate tools. The most prominent example of this approach is the Java Modelling Language (JML) [36], with a variety of tools described in [187]. Furthermore, work is currently underway to bridge UML and JML by translating OCL into JML [188]. A consequence of this work is to allow UML to exploit the variety of available JML tools for formal reasoning about UML specifications [189].

From the above discussion, it becomes clear that UML with each formal component OCL is currently the best choice with respect to balancing formality and usability of specification-based approaches to software reuse. As a result it is the specification language of choice for our component type model.

**Formal behaviour specification**

In the previous section we selected UML/OCL as our behaviour specification language, as it provides a good balance between formality and usability. In this section we examine how particular formal constructs can capture different aspect of component behaviour. We start this discussion with the constructs used in contract-based
programming [149], namely pre-/post-conditions and invariants. Using pre-/post-condition is widely used technique for capturing the behaviour of software components with respect to their operations. Its popularity is demonstrated by the fact that besides contract-based programming approach, they have also been incorporated as constructs in programming languages like Eiffel [215] and they have also been included as stereotypes of OCL. Moreover, in the area of component selection in software reuse, pre-/post-conditions have been used to define a variety of ways in which function specifications may be matched to retrieve components with appropriate behaviour [166]. Furthermore, these specification matches have also been combined with a variety of ways in which the function signatures may be matched [167]. This work has also been extended to cover modules, software components containing a number of functions, in which case the invariants are also taken into account. As these matches have been devised with component reuse in mind, they could provide an appropriate basis for the definition of semantic type relationships in our component type model. As a result, we require that the behaviour of component types is at least specified in terms of pre-/post-conditions and invariants with respect to their operations.

However, it is widely recognised that these constructs are not able to capture the full spectrum of potential component behaviour. For example, although invariants can be used to express static constraints on the component state, they cannot express constraints on the transitions between states. This is quite serious limitation with respect to behaviour subtyping of object-oriented systems. To address this problem Liskov and Wing in [197] introduce history constraints. Since, as we have seen in section 2.3.4, state-of-the-art component platforms are based on some kind of object model, we take the view that descriptions of component behaviour should include in addition to pre-/post-condition and invariant also history constraints.

Another important issue to consider with respect to the specification of component behaviour is the so called callback problem [7]. This problem refers to the fact that using declarative specifications of component behaviour it is not possible to guarantee that the component encapsulation boundary is not breached in the presence of callbacks or self-recursive calls exposing inconsistent component states. Szyperski was the first identified this problem. The problem is clearly demonstrated in the case of the
observer pattern [114]. In the observer pattern, when a subject notifies its observers for a state change, the observers as part of their reaction may request the new state by making a callback to the subject. Depending on whether the subject has completed the update of its state or not, the observers may be to an inconsistent internal state, i.e. a state that external entities should never have seen. Szyperski provided a solution to this problem by introducing test variables in the specification of method preconditions, which prevented any callbacks, while the update of the component's state was still in progress. However, his solution was very restrictive and fairly ad hoc. This problem can be tackled in a lot more flexible and structured way with the use of operational specifications [216]. However, some researchers argue that declarative specifications of behaviour are preferable since they are more abstract and easier to understand [217, 218]. Leavens and Dhara provide a solution to the callback problem in [5] by extending JML [36] to include model programs. Their solution seems like an interesting compromise that we need to adopt in particular when specifying the behaviour of events.

In general a number of other extension have been proposed to address the limitations of pre-/post-conditions to capture particular aspects of behaviour, either with the introduction of additional constructs, or with the introduction of more expressive logics. However, we need to keep in mind that the complexity of the behavioural specification has also a significant impact on the tractability of the deduction process required to establish any relationship between them. Even the use of first order logic in describing the predicates of the behavioural specifications, which is the most common approach, makes the deduction process in most but the simplest cases undecidable. Thus it cannot be fully automated. A number of techniques have been proposed to improve the performance of theorem provers, in terms of both the number of proofs they can perform, and the time in which they perform them. These techniques range from feeding of the theorem prover with a number of relevant lemmas and how these should be determined, to using alternative proving techniques for the same proof. Although, this whole issue is the subject of current research, according to Meyer [185] progress looks promising.

At the same time, Hussmann's in [8] provides a list of seven theses for the practical usefulness of formal specifications of component behaviour that relieve some the burden for proofs. These theses are:
1. "Components cannot be considered in isolation.
2. Full formal specification of business domain semantics is too much.
4. Critical component co-ordination issues require precise specifications.
5. Non-functional requirements are important.
6. Formal specifications may provide a flexible mechanism for browsing/viewing component configurations.
7. Formal specification shall be interpreted by machines and still be readable for human beings”.

Commenting on these theses, the first one refers to the fact that components are always constructed in reference to a particular domain and the component specification uses the terminology defined as part of that domain. The second one refers to the fact that the vocabulary for each business domain has usually been well defined and trying to formally specify it is unnecessary. The third one makes the point that in most cases it is too much to specify the semantics of the underlying component architecture. Although, this is true in most case, attention should be brought to the problems of architectural mismatch first identified by Garlan et al. [103]. The fourth thesis stresses the importance of specifying these aspects of the components that are crucial about their correct configuration instead of certain functions of the business that might be quite clearly understood. The fifth thesis stresses the importance of extra-functional requirements, which is an area that requires most of the attention since at the moment most of the work in this area is quite immature. In relation to the sixth principle there is some work on this area, which has produced some interesting results, we will analyse these results later on. Finally, the seventh thesis is a more general one and relates to the criticism about the usability and the cost effectiveness of formal specifications we referred to above.

**Concluding remarks**

In conclusion, specification based retrieval techniques are the most appropriate as the basis for SECT. The main issue regarding these techniques is the level of formality of the specifications. Although, making formal the semantics of the specifications eliminates ambiguity it also makes them difficult to use. A number of approaches are trying to balance these largely conflicting requirements. We believe that UML with its formal
component OCL strikes a reasonable balance and we adopt it as the language for the
description of component behaviour. A particular feature of UML/OCL is that they
provide mechanisms for their extension. For this reason we identify a number of formal
specification constructs that are not only essential for the description of component
behaviour, but also allow us to take advantage of software reuse techniques like signature
and specification matching. Moreover, as there is a trade-off between the expressiveness
of the behavioural specifications and the tractability of the deduction process for
establishing relationships between them, we decided to not consider any other constructs.
These constructs are pre-/post-conditions, invariants, history constraints and model
programs. Finally, we think that it is important that our approach should takes into
account Hussmann’s seven theses.

3.3.2. Object interoperability

In general, “interoperability is the ability of two or more software components to cooperate despite differences in language, interface, and execution platform” [219].
Interoperability concerns are usually addressed at three levels [40]:

1. Signature level (names and signatures of operations),
2. Protocol level (relative ordering between exchanged messages and blocking
conditions), and
3. Semantic level (the “meaning” of the operations).

Although it may appear that the three levels of interoperability are independent,
this is not really the case. In fact, Canal et al. argue that “…interoperability should be
studied in general at the semantic level. The problem is that this semantic level covers a
very broad set of issues and there is not even a consensus on the full scope of what those
issues are” [201]. As a result, the boundaries between the three levels are quite difficult
to discern. At the same time, though, “mixing all … [levels] within one component
specification will produce too large, complex and brittle specification to be of any
interest” [201]. Therefore, dealing with each level of interoperability separately allows us
to specify components in a modular way, which leads to separation of concerns and to
support of more complex and changing requirements. This view of separating the
different levels of interoperability is further reinforced, by the fact that the techniques
used to deal with interoperability issues at each level are quite distinct, e.g. [167, 224] for signature level, [200, 201] for protocol level, and [27, 166, 193, 173] for semantic level. For these reasons we take the view that each level of interoperability should be dealt with separately.

The problem of defining the boundaries between the three levels still remains. For this reason we take a closer look into the related literature. In order to define the boundary between protocol and signature interoperability Wegner offers the analogy between connecting electrical appliances to the power and connecting software components together [219]. In both cases, we have to deal with an interoperability problem that has two aspects a static one and a dynamic one. The static aspect deals with the problem of fitting the appliance’s plug to the power socket or in the case of software components supporting corresponding interfaces. The dynamic aspect deals with the problem of voltage and current that the socket provides and the plug requires or in the case of software components the correct usage of the respective interfaces. In this sense, signature level interoperability deals with the static aspects of the problem, while protocol interoperability deals with its dynamic aspects. So, according to Wegner interoperation at the protocol level refers mainly to the preservation of temporal properties like order constraints on operations or coordination of inputs from multiple input streams, etc.

Canal et al. adopt a similar separation and suggest the inclusion in protocol descriptions of just the information about the components’ interaction, keeping away as much as possible the “functional” aspects of components [201]. But, they warn that this kind separation between the computation and coordination aspects is often subtle. Han also adopts this kind of separation and combines it with a constraint-based approach [221]. The constraint-based approach assumes that “if there are no interaction constraints specified for a component, any interaction sequences and states are allowed subject to the component’s signature”. Interaction constraints define the interaction protocols of the component and “provide explicit guidance about how to interact with the component. Observing these constraints is necessary to avoid exceptions, errors and unpredictable behaviour and to ensure proper use of the component in a given context” [221]. Despite the warning of Canal et al. we adopt the same kind of separation. In fact, we push Han’s
view to the extreme by considering protocol level specifications as guidelines for the
correct usage of the components and as such although we provide placeholders for their
description we do not consider them as part of our component type model. The
placeholders play also the role of extension points allowing the future extension of our
component type model to include protocol description and compatibility, in a similar way
to Iribarne et al. [202]. Another reason for ignoring protocol interoperability is that that
protocol compatibility can be checked also at runtime. This involves the interception of
all exchanged messages between components and the verification of their correctness
with regard to the current state.

Finally, with respect to semantic interoperability, although it was initially devised
to have a very broad range [173], we observe that when authors refer to “semantics”, they
really mean “operational semantics” or “behavioural specifications” and propose
formalisms such as pre-/post-conditions, temporal logic and refinement calculus to deal
with interoperability issues at this level [220]. This view is similar to the one we took in
section 3.3.1. In fact, we take a more specific view and we selected the pre-/post-
condition formalism for the behavioural specifications.

3.4. Concluding remarks

In this chapter, starting from an examination of the service location and discovery
problem in networked and open distributed system, we first showed that current state-of-
the-art component platforms with their support for some form of references and home
interfaces take care of the location problem. Then, we identified service trading as the
most appropriate basis for a component location and selection service. Following a close
examination of service trading, we found that its syntactic nature is a serious problem for
its adoption. In order to overcome this problem, we introduced the notion of semantically
enhanced component trading (SECT), a new kind of trading that is based on a component
type model incorporating notions of semantic types and type relationships. Our analysis
of SECT with respect to component retrieval in software reuse and object interoperability
led to the identification of a number of defining characteristics for its component type
model. These characteristics are summarised below:
• The types of the model should include descriptions of both signature and behaviour.

• Interfaces play a central role in the description of types, but the notion of interface needs further clarification.

• Type descriptions should deal adequately with all three levels of interoperability, but it is preferable to deal with each level of interoperability separately.

• Descriptions of type behaviour should be specification-based with a balanced approach to formality. UML with its formal component OCL is a promising candidate for this role.

• Behavioural specifications addressing protocol level interoperability can be considered as rules of correct component use and as a result the component type model does not have to fully utilise them.

• Behavioural specifications addressing semantic level interoperability are described in a contractual way using preconditions, postconditions, invariants and history constraints.

• In behavioural specifications addressing semantic level interoperability we should always keep in mind Hussmann's theses [8].

• The component type model should not only support the description of type behaviour, but should also define semantic notions of compatibility that consider these behaviour descriptions. Moreover, it should be flexible enough to provide various relaxed forms of conformance.

• The component type model should be extensible allowing the introduction of new type concepts and the definition of new type relationships.

In the following chapter we define a type model including all these characteristics.
Finally, with respect to remote interpretability, although initially described as a generic framework for remote interpretation, the tool was implemented and instantiated in a specific context. Programming languages such as Python and Java have been used to deal with these languages. As a result, the integration of the remote interpretability framework with the specific language tools has been facilitated.

At the system level, the remote interpretability framework has been integrated with a specific language tool.

Chapter 4. SECT: component type model

In the previous chapter we introduced the notion of semantically enhanced component trading (SECT), a new kind of trading that is based on a component type model incorporating notions of semantic types and type relationships. We also identified a number of defining characteristics for its type model. In this chapter, starting from these characteristics, we carry on to fully specify the details of the type model. Our presentation in this chapter exhibits certain similarities with the approach that Indulska et al. followed in [1] for the specification of a service trader for ODP. However, in our case, the emphasis is on component types and component type compatibility relationships. We start with a brief overview of the main characteristics of our component type model in section 4.1, followed by a detailed examination of the construction of the type model in the subsequent sections.

More specifically, in section 4.2 we define the main concepts of our type model (i.e. components, services, actions, their interfaces, and component and service properties), specify its general characteristics and justify in detail all our choices. This is followed by a discussion of the naming scheme adopted for the types in section 4.3. In section 4.4 we specify how each of the types of our model are described in a top-down fashion starting from the component types and ending with the basic types. Section 4.5 defines the relationships between the types of our model focusing on type compatibility/conformance relationships. These relationships are defined across a number of different dimensions with particular emphasis on structure-based and semantics-based relationships. Section 4.6 discusses the kinds of polymorphism supported by the type model and introduces a number of inheritance relationships. Finally, in section 4.7 we close the discussion of our type model with the introduction of the notion of type definition domains and we show how they are incorporated in type descriptions and what their effects are on the type relationships defined earlier on.

4.1. A brief overview of the component type model for SECT

The component type model incorporates the usual basic types, type constructors, and data types. Moreover, it introduces types for actions, services and components. More
specifically, actions are an abstraction of operations and notifications, while services 
comprise of a set of actions, and components comprise of two sets of services, provided 
and required ones.

Its main characteristic is the semantic nature of component, service and action 
types. In other words these types specify both their structure and their behaviour. The 
structural specifications of the types take the usual form. More specifically, in the case of 
action types it describes the name, the potentially empty list of argument types, the 
normal termination type and the potentially empty list of exceptional termination types. 
In the case of service types it describes the structure of its constituent action types. While, 
in the case of component types it describes the structure of its provided and require 
service types. On the other hand, the behavioural specifications of the types in the case of 
action types take the form of pre-/post-conditions and possibly model programs. However, in the case of service and component types the behavioural specifications 
consist of two parts, namely a semantic constraint and a protocol constraint part, 
addressing semantic and protocol level interoperability concerns respectively. The 
component type model is primarily concerned with the former kind. Semantic constraints 
are described in reference to either a service behavioural model expressed in a model- 
based manner, or a component behavioural model defined as the aggregation of the 
service behavioural models of its provided and required service types. It is interesting to 
note that in the case of service types in particular the behavioural specification includes 
both invariants and history constraints.

Another interesting characteristic of the component type model is the annotation 
of service and component types with property types. These property types are either 
atomic or composite and support the specification of extra-functional properties of the 
service or component type they annotate. In order to allow the specification of multiple 
service and component types by varying only their extra-functional property annotations 
the component type model also introduces interface types. However, in addition to the 
usual syntactic action (i.e. action signature), service and component interface types, it 
also introduces service and component behavioural interface types. All the above 
elements of the component type model are examined in detail in section 4.2 and are
summarised in the OMG Meta-Object Facility meta-model (see section 2.3.2) in Figure 4.1, with the main characteristics highlighted in colour.

![Figure 4.1. A Meta-Object Facility meta-model of the SECT type model.](image)

As well as introducing the above kinds of type the component type model also introduces a variety of type relationships. The emphasis is on type compatibility/conformance relationships that are useful during the trading process. These relationships are defined within a unified framework that takes into account the primary elements of type definitions, namely type names, structure and semantics (i.e. behavioural specifications). Within this framework each element defines a class of type compatibility/conformance relationships. In the case of named-based relationships, the component type model defines basic name compatibility in terms of name equality. While, for the
structure-based type relationships it builds upon previous work on function and module signature matching [167, 191] and extends it to cover component interface types, and contra-/co-variance type compatibility/conformance based on the modes of action parameter types. At the same time, the component type model combines previous work on specification matching [166, 191, 194], behavioural subtyping [197, 193 27, 5], reuse ensuring specification matches [195], and program refinement [196] to build a number of alternative semantics-based type compatibility/conformance relationships. Furthermore, the component type also defines syntactic-based type compatibility/conformance relationships as combinations of name-based and structure-based ones. All the above kinds of type relationships are elaborated further in section 4.5.

It is interesting to note that the component type model rigorously defines how all its types may be described and provides a template for the rigorous description of type relationships. It also describes how UML/OCL can be extended to support the specification of service and component behavioural models.

Besides the type and the type relationships the component type model also introduces the notion of type definition domains. On one hand these domain can be used to standardise extra-functional property contracts [25] and composite property types [37]. On the other hand, they can be used to standardise service and property types as well as vocabulary used in particular application domains.

Standardised types are also supported by the type naming scheme adopted by the component type model. The scheme guarantees the uniqueness of type names even in the cases of trader federations. More importantly, though, it classifies type names as either private, i.e. known only to the type manager in which they introduced, conformant, i.e. standardised within a particular application domain and universal, i.e. known to all type managers. The latter are primarily used for the definition of basic types and type constructors.

Finally, the component type model also supports both parametric and inclusion polymorphism for service and component type, service and component behavioural interface types, and service and component interface types. At the same time it supports inheritance for these types as a mechanism for the incremental definition of types.
4.2. Types

The starting point in determining the types that our component type model includes is the type models used by service traders.

4.2.1. Service trading type models

An examination of service trading type models identifies some common characteristics. All these models include a set of primitive types. These usually include both basic data types and some additional types with special meaning. The basic data types refer to common types like integers, character, etc., while examples of additional types include interface references (InterfaceRef) and Any types in CORBA [24]. Most models will also include some built-in type constructors that allow the creation of composite types like records, unions, sequences, etc. Most type models adopt an object-based notion of services. For example in ODP “a service is functionality provided at the interface of an object” [11]. This approach takes advantage of the encapsulation and abstraction properties of the object model.

Particular emphasis is put on the role of interfaces that define the encapsulation boundary of the object and the respective service. As a result most of these type models include a notion of interface type, which describes sets of operations, akin to method invocations in the object model. In fact, some type models include multiple notions of interface, e.g. ODP [11] includes operational, stream and signal interfaces, for request-response, data flow, and single message modes of interaction respectively. The emphasis on interfaces is particularly apparent in the quite common use of IDLs, which provide a programming language independent set of primitive types and type operators. These primitive types and type operators are consequently mapped to the various supported programming languages.

It is interesting to observe that although interface types are described in terms of operations, quite often the notion of operation type is not part of the type model. Moreover, although this type models include a notion of service type, which is usually defined in terms of interface types, it is surprising to note that service types are not explicitly described as part of the type model. The absence of operation types and explicit description of service types is perceived as a weakness of the type models. To address
this weakness, Indulska et al. [1] proposed a type model for ODP that enhances the standard with the inclusion of both explicit description of service types and operation types.

Other general characteristics that the service trading type models have include polymorphism in both parametric and inclusion form and inheritance. Parametric polymorphism is usually supported through Any types, which can represent any other type of the type model, while inclusion polymorphism is quite often linked with some form of inheritance. Finally, inheritance is quite often supported both at the interface and the service level.

From the above discussion it should be clear that the type models for service trading are quite comprehensive and as a result it is only sensible to incorporate into our component type model their common characteristics. Hence, in summary our component type model will include a set of basic types, some built-in type constructors, some notion of service and operation type. It will follow an object-based notion of service, with interfaces describing the service encapsulation boundary. It will also provide support for both types of polymorphism and inheritance at the service and interface level.

4.2.2. Component types

So far we have already identified a preliminary set of types for our type model, but the issue of how to introduce a notion of component type remains open. In the definition of architecture description languages (ADLs) we encounter notions of component types. For this reason we start this section with an examination of the related literature.

ADLs describe system configurations in terms of components, the primary computational elements and data stores, and connectors, the elements of communication and coordination between components, along with constraints on their usage [12]. This is particularly underlined by Tracz's definition of an ADL as consisting of four “C”s: components, connectors, configurations, and constraints [13]. Regarding the role of types in ADLs, Shaw et al. [14] assert that types are an important property that an ADL should exhibit, meaning that in an ADL both components and connectors must be typed. Garlan [15] further reinforces this view, when he says that an architectural style can be viewed as
a system of types, where the architectural vocabulary (components and connectors) is defined as a set of types. In general, ADLs specify the characteristics of component and connector types through a description of their respective interfaces, which act as their encapsulation boundary. Medvidovic and Taylor [12] recognising the central role that interfaces play in ADLs and consider them as a separate dimension of their ADL taxonomy. Therefore, in order to determine the characteristics of component types, we need to examine what kind of characteristics their interfaces describe.

**Interfaces in ADLs**

Before we go any further we should remind the reader that in section 2.1 we examined a number of definitions for the notion of component and we selected the one proposed by Szyperski (see Definition 2.1). In this definition interfaces have a prominent position. Furthermore, the definition requires that component interfaces must be contractually specified, while components should only have explicit context dependencies. According to Jonkers [6] "components use interfaces to make their context-dependencies explicit". Luckham et al. characterise this kind of interfaces as contextualised [3]. In fact, they describe five varieties of specification that an interface in an ADL should contain:

(i) A list of provided features,
(ii) A list of required features,
(iii) A specification of the behaviour of provided features,
(iv) A specification of the behaviour of required features, and
(v) A specification of any interaction between two or more features.

Medvidovic and Taylor [12] take a similar position when they say, "an interface defines computational commitments a component can make and constraints on its usage".

Although almost all ADLs support a similar notion of component interface, there is a lot of variation in both the terminology used to describe interface points (referred to as features, ports, constituents, players or services) and the kinds of information they describe. For example, Luckham et al. [3] refer to features, architectural elements of a component (e.g. a function, a port or an action), while Medvidovic and Taylor [12] refer to interaction points. Both of these approaches take the view that each component has a single interface describing all its associated features.
Garlan et al. [17] adopt a different view according to which a component may have multiple interfaces, each defining a point of interaction. They refer to every such interaction points as ports, which can either be a single procedure, a collection of procedures that must be invoked in certain specified order or an event multicast interface. This view is similar to the one adopted by the CORBA Component Model (CCM) [18]. The idea of grouping related interaction points/features into services is also present in Rapide [19], and is utilised in plug and socket architectures where dual services are connected by service connections in order to deal with scaling issues in ADLs [3]. The same idea is behind the concept of interface suite, a group of mutually related interfaces [6].

Han takes a slightly different view in that it considers each component having a single interface, but at the same time introduces the notion of multiple roles that a component can play each capturing part of the whole interface [20]. We should note here that Han in his work does not specify a new ADL. Instead he combines elements from object orientation analysis and design with elements of software architecture description languages in an attempt to define a comprehensive interface definition framework for software components.

From the various approaches described above, the grouping of related component interaction points or features allows for more cohesive and manageable specification of component types. For this reason, we adopt this approach and we consider components consisting of a number of services. These services are cohesive groupings of related interaction points of the components. As a result, component interfaces describe the set of services that a component is comprised of.

Variations on the interface theme

Another point of variation with ADLs is how they distinguish between incoming and outgoing interface points, or input and output ports, or provided and required services. As we mentioned above, the contextualisation of a component’s interface requires that both provided and required features must be described. Not all ADLs include constructs that allow the explicit distinction of provided and required features. Moreover, even if the language provides the necessary constructs there are still variations in the semantics associated to them. For example, Han [20] specifies required properties,
operations and events (using the keyword require) and provided properties and operations (using the keyword provide) as part of each individual role. While in CCM [18] the services are divided into provided (using keywords like provides, emits and publishes) and required ones (using keywords like uses and consumes), where every service has its own interface. In order to reinforce the advantages of viewing services as cohesive groups of features, in our component type we follow a similar approach to CCM. Subsequently, we characterise services as either provided or required ones.

The two examples we used above highlight the issue of events, which is another point where variation between different approaches can be observed. In certain approaches events that components produce or consume are not treated separately from operations they require or provide, both are described in the same way. These approaches usually introduce some notion of action, an abstraction over the type of interaction between components (synchronous or asynchronous, point-to-point or multicast, etc.). While in other cases events and operations are described separately (usually using different keywords). Catalysis [21] is an example of the former approach, while Han’s approach [20] and the CCM [18] are examples of the latter. An interesting observation regarding Han’s approach and the CCM is that in the former the description of events is different from the description of operations, event descriptions just require the event’s name, while operation description require the full operation signature. While in CCM both events and operations are described through a similar type of interface description according to CORBA IDL, but the semantics associated with each type differ (e.g. a publishes interface implies event multicast, while an emits interface implies event unicast).

The issue of events is not just a matter of semantic differences in language construct labels. As we discussed in section 3.3.1 events can introduce problems in behavioural specifications, particularly in how these specifications deal with the issues of callbacks and re-entrance. Putting these considerations aside for a moment, we take an approach similar to Catalysis and treat both operations and notifications the same. We achieve this through the introduction of an action abstraction that services are comprised of and all actions are described in exactly the same way. Accordingly, we are going to
deal with the issues that event specifications raise in our behavioural model of actions. In summary, are component types consist of a set of provided and required services, which in turn consist of a set of actions.

4.2.3. Contextualisation of component and service interfaces

As we mentioned above, according to Luckham et al. [3] the contextualisation of a component's interface requires both a behavioural specification of each provided and required feature and the specification of any interaction between interface features. This requirement raises the question of what makes an interface well defined or in other words what exactly should be included in interface descriptions. This question has received a lot of attention in recent years both from the point of view of interface definition and architecture description languages.

From the point of view of IDLs, this interest stems from the increased emphasis in encapsulation, abstraction and information hiding. This increase is fuelled by the success of the object-oriented model that emphasises these same properties and is pushed even further by distributed object platforms (see section 2.3 for a presentation of their characteristics) that introduce standardised IDLs to achieve interoperability. These same properties are also a requirement in component software. In all cases the goal is to ensure that interface descriptions provide adequate information to avoid any runtime errors. Depending on the context in which we consider this goal the focus could be on different types of runtime errors and consequently the way interfaces are described would be different.

A number of extensions to traditional IDLs have been proposed to address the different types of runtime errors. Jacobsen and Kramer in [16] provide an extensive survey of such extensions, which include behavioural extensions, real-time constraints, quality of service attributes, interaction protocols, and synchronisation and co-location constraints. It is easy to see that all these extensions relate to different aspects of interoperability at one of the three levels that we discussed in section 3.3.2. Since, it is preferable to address each interoperability level separately, in our component type model we reserve the notion of interface to refer only to signature level interoperability aspects.
This approach is inline with traditional service trading, where interfaces only refer to syntactic aspects.

In an attempt to address similar considerations ADLs attach behavioural specifications to each interface element, whereas the interaction between interface elements are addressed by the introduction of a constraint part to the interface. In general, though, their notion of constraint has a more general use. In ADLs “a constraint is a property of or assertion about a system or one of its parts, the violation of which will render the system unacceptable to one or more stakeholders” [22]. The specification of constraints is supported either through the use of a separate constraint language or through constructs of the ADL [12].

There is a lot of variation in the kinds of constraints different ADLs support. These constraints include invariants, extra-functional attributes or properties, which in some cases are divided into required and optional, and protocols of interactions. It is worth noting that in addition to the explicit specification of constraints, most ADLs constrain the usage of components implicitly by requiring adherence to the interface description, the behavioural specification of interface elements and the semantic model underlying the language.

Han [20] takes the notion of constraints from ADLs and introduces it in his interface definitions. He uses constraints both as part of the component’s interface description and the component’s role specification. The former constraints are divided into two types, those concerning individual interface elements and those concerning the relation between different elements. While the latter concern the relationship between different roles that the component plays. We follow a similar approach and we associate constraints with both component and service interfaces. In this way besides that syntactic notion of interface of traditional service trading we also introduce a notion of contextualised interface, which we call behavioural interface. We can view our notion of behavioural interface as an extension of traditional service trading interfaces similar to the various extensions examined by Jacobsen and Kramer in [16].

In order to further separate the various levels of interoperability, we separate, the constraints we associate to component and service interfaces into two parts. One part contains constraints on the relationship between individual component and service
elements and one part contains all other kinds of constraints. The former kind of constraints mainly refers to the ordering and sequencing of operations calls or message exchanges. It describes the protocol that should be followed when interacting with a particular component, i.e. to protocol level interoperability information. As we mentioned at the beginning of this chapter protocol interoperability constraints can be considered as guidelines for the correct usage of the components or services. Therefore, although we provide placeholders for their description we do not consider them further in our component type model. This is in contrast to the latter kind of constraints, which we consider as part of the behavioural specifications of component and service interfaces, and provides semantic level interoperability information. This kind of constraint is an integral part of our component type model as will be clear when we discuss the behavioural model of our types later on.

In summary, our component type model includes two kinds of component and service interfaces: plain interfaces, and contextualised or behavioural interfaces. The former only contain signature level interoperability information, whereas the latter contain interoperability information for all three levels. In behavioural interfaces, we achieve the separation of the information for each level of interoperability with introduction of two separate constraint parts, one containing protocol and one containing semantic level interoperability constraints. The latter type of constraints is further elaborated in the behavioural model of our types.

4.2.4. The issue of extra-functional properties

Before we describe the behavioural model of our types we should first discuss the issue of extra-functional properties or illilities. The problem with respect to extra-functional properties as Iribarne et al. [37] point out is that despite their widely recognised importance, most proposals for documenting components fail to take them into account. The work by Han [20] is a notable exception where the description of illilities has a prominent place in interface descriptions. They define an additional dimension of interface descriptions that cuts through all other element descriptions. The main reason for not considering extra-functional properties is the difficulty in their specification and analysis. This difficulty according to Chung et al. [38] is due both to their subjective and
relative nature, and the potentially complex interactions between them. This means that the description, interpretation, evaluation and importance of extra-functional properties may vary depending on the particular domain into consideration. While at the same time, certain requirements can help or hurt the achievement of others. As a result, it is quite difficult to come up with a general scheme for describing them.

For this reason, in service trading illities are just described through properties, which are plain name value pairs. These properties are either associated to the service as whole or to particular elements of a service, e.g. to particular operations. A typical example of this is the fact that service types in CORBA [24] are described by an interface and a set of property types associated with it. This kind of approach is very simplistic and quite difficult to manage. For example the similarity between certain extra-functional properties may be lost because they are specified in a different way, while the specification of multiple different illities may lead to an explosion of properties.

Iribarne et al. in [37] propose that a more manageable approach to the description of extra-functional properties can be identified if it is integrated within a requirements engineering framework. In their proposal they use the NFR framework [38] for this purpose. NFR views each extra-functional requirement as a set of goals that can be successively decomposed into refined sub-goals until they can be related to functionalities that implement initial extra-functional requirements. During the decomposition process any interactions between sub-goals of different requirements can be documented. In this approach the description of extra-functional characteristics is still done by associating properties to the functional elements of the component description. These properties in accordance to the NFR view are either individual ones or compositions.

The use of property compositions introduces some initial structure to the set of properties associated to service and component types. Frolund and Koistinen [25] structure properties even further with the inclusion of contract types into their quality of service specification language (QML). QML supports the definition of various QoS contract types, each representing a particular QoS aspect, e.g. reliability or performance. A contract type defines the dimensions that can be used to characterise the particular QoS aspect. Each dimension also specifies its range of potential values. Contracts are
instances of contract types and represent particular QoS specifications. QML profiles associate contracts with interfaces, operations, operation parameters and operation results. QML separates the specification of functional and QoS characteristics of services and component types, thus allowing the association of different QoS profiles to the same type or the sharing of QoS profiles between multiple types. Although, QML is only aimed at QoS properties, we believe that the approach could be extended to cover other extra-functional properties, hence providing a generalised notion of extra-functional contract. This extension of QML would of course require further investigation, which is out of the scope of this thesis.

In our type model we follow a similar approach to CORBA and associate property types with component and service types. In fact, we take the view that the addition of property types is the only difference between a component or service type and its respective behavioural interface. At the same time, we support a more manageable approach to properties by allowing reference to particular extra-functional property contract types similar to the ones specified in QML and property compositions similar to the ones described by Iribarne et al. [37]. We also allow the standardisation of particular contract types and property compositions, which can then be reused by a number of component and service type descriptions. Following this approach we separate the specification of extra-functional properties from the specification of the behaviour of the various interface elements and the component and service constraints addressing semantic level interoperability issues.

4.2.5. A model of service and component behaviour

Thus far, we have clarified the scope of the behavioural descriptions, attached to component and service interface elements during their contextualisation. We can now examine the model of behaviour on which they are based. The construction of the model is done in two stages. First we adopt an object-based view of services, which combined with a model based approach to specification constructs our model of service behaviour. Then we introduce another level of encapsulation and view multi-service components as compositions of objects. We decided to adopt an object-based model of services, because according to Christiansen et al. [4] its advantages in terms of encapsulation make it the
most commonly followed approach. As its name implies the foundation of the object based service model, is the notion object. An object has unique identity, state, and behaviour that depends on its internal state [4]. The interface of an object is the only way to interact with it and observe its behaviour. The interface describes a set of actions that determine the behaviour of the object and affect its state. Remember that according to our type model we can view actions as abstractions of the way in which objects interact with each other. A service is functionality provided at the interface of an object.

In model-based specifications, as the name implies, the specifier defines a system's behaviour by constructing a model of it [26]. The model of the system's behaviour is described in terms of mathematical structures such as tuples, relations, functions, sets and sequences. This kind of specifications maps nicely to the object model. We can model the externally observable states of an object as tuples of the values. These values map to the values of the attributes of the object. Then, we can specify the behaviour of a service in terms of its actions, where each action when activated in a particular state produces a new state. These two states are what the pre- and post-conditions respectively describe. As we discussed in section 3.3.1, the completeness of these specifications requires in addition to pre- and post-conditions the specification of invariants and history constraints (see also [27]). Consequently, each service needs to define a model for its observable states and specify its behaviour by describing the pre- and post-conditions of each of its actions and its invariant and history constraint.

However, such an approach is not adequate in a component software setting. As we pointed out in section 3.3.1 the specification of action behaviour, as a black box is too abstract in the case when events or notifications are produced as part of the action [28]. This issue is the cause of the callback problem as identified by Szyperski [7]. We can address this issue by following the approach suggested by Leavens and Dhara in [5], where the behavioural specifications of services are enhanced with the addition of model programs. This approach allows specifications to expose the fact that callbacks are part of the action and to constrain the behaviour of the callbacks. As a result, our behavioural model of services besides pre- and post-conditions allows also the description of model programs as part of action specifications.
In our type model we consider components as a set of services provided in context (provided services), where the context is specified as the set of services the component depends on (required services). Taking into account the above discussion we can say that a component provides functionality at the interfaces of a set of objects and requires certain functionality provided by other components, or in other words a component consists of a set of interrelated services, which provide functionality at the interfaces of a set of interconnected objects. Since according to Definition 2.1 components are the units of composition and deployment, their interface acts as an encapsulation boundary for the included services and consequently the objects providing the functionality of the services can only be used as a whole. So, our behavioural model of components only describes constraints on the set of provided and required services. Note that these constraints restrict the behaviour of the services both in terms of possible states and in terms of the possible state transitions. In this sense are similar to both the invariants and the history constraints of services.

Following a model-based approach for the specification of component behaviour raises some challenges regarding the behavioural specifications of required service types. If the behaviour of required services is specified in exactly the same way as provided ones then the specifier of the component needs to have knowledge of the internal behavioural models of any service that might provide the required functionality. This requirement is fine as long as there is some agreed common basis between the specifier of the required service and the specifier of the provided one. In the general case it seems unreasonable to expect the specifier of the required service to know the behavioural model of any type that might provide the required functionality. Medvidovic et al. first identified this problem in [29]. They suggested as a solution the introduction of a new type, namely STATE_VARIABLE, which is assumed to be a supertype of all the types of the type system. This new type is used as a placeholder in the predicates of the specification of the behaviour of required services. In our behavioural model for components we adopt a similar approach and introduce a type called MODEL_VARIABLE to play the role of placeholder. The details of how exactly this type is used in behavioural specifications and the reasons why we selected a different name for it will become apparent later on.
Summarising our model of behaviour, we use model-based specifications of behaviour with services as the central focus. Each service specifies a model of its externally observable states. The behaviour of the service is specified in terms of its state model through invariants, constraints on the set of permissible states; history constraints, constraints on the permissible state transitions; and the specification of the behaviour of its actions, which cause the state transitions. The specification of the behaviour of actions is in terms of pre-conditions, the set of states on which the action can be activated; post-conditions, the state produced after the action is carried out; and possibly a model program specifying the effects of events or notifications. Component behaviour is specified through the specification of its provided and required services, and constraints relating their respective state models. Finally, required services specify their state model using MODEL_VARIABLE placeholders instead of concrete state variables.

4.2.6. The types of the component type model

We can now list the various kinds of types of our component type model. Note that these refer to meta-types in the language of the meta-object facility [30], or kinds, "types" of types, according to Cardelli [31].

- **Component Types.** A component provides packaging for a set of services and in this context a component type describes a particular way in which a set of services can be packaged within a particular context. The component type specification describes the details of the packaged services, provided services, and the context, required services.

- **Service Types.** A service represents a cohesive piece of functionality. This functionality is provided through a set of actions (either operations or notifications). The set of actions and the specification of the behaviour describe the functional properties of the service, while a set of properties allows the description of its extra-functional properties, including QoS related characteristics.

- **Action Types.** An action is an abstraction of the interaction between services. Actions may have a number of parameters including none, and a number of terminations including a normal termination and exceptional ones. Normal
terminations may return a result, while exceptional terminations maybe parameterised. Each action type specifies the signature, parameters and terminations, and the behaviour of the actions. Action types can only be specified within the context of a service type since their behavioural specifications are in reference to a service state model.

- **Property Types.** Properties are name value pairs. Properties can be associated with service and component types. As a result, property types are classified into service and component ones, which describe different kinds of characteristics. Property types are also classified as common, domain-specific and component/service specific depending on the scope of their specification.
  
  - **Common property types** have a global scope and can be used as part of any component/service type specification.
  
  - **Domain-specific property types** are standardised within a particular domain and can be used as part of any component/service type specification aware of the domain. An example of a domain for domain-specific property types could be a contract type similar to the ones suggested in [25].
  
  - **Component/service specific property types** are introduced as part of a component/service type specification and their scope is limited.

All property types are either atomic, a single property type, or composite, a set of property types describing an extra-functional property contract. The main difference between component and service property types is that the former can only be atomic while the latter can either be atomic or composite.

- **Component Behavioural Interface Types.** The behavioural interface of the component includes everything the component type specification includes with the exception of the property types. They also are contextualised component interface types.

- **Service Behavioural Interface Types.** The behavioural interface of the service includes everything the service type specification includes with the exception of the property types. They also are contextualised service interface types.
• **Action Interface Types.** Action interface types only refer to the signature of an action specification.

• **Service Interface Types.** Service interface types refer to the set of action interface types of the respective action types included in the service type specification.

• **Component Interface Types.** Component interface types refer to the set of service interface types of the respective service types included in the component type specification.

• **Basic Types and Type Constructors.** For reasons of simplicity we assume the same set of built-in basic types as in CORBA IDL [24]. The only difference is the replacement of the generic CORBA Object References by Component Interface References and the generic CORBA Interface References by Service Interface References. Note although in CORBA IDL built-in basic types also include the Any type, we treat Any type as a separate basic type. We too assume the Any type to be a supertype of all built-in and constructed basic types, but we exclude constructed basic types using the Any type as a base type. We take this approach in order to avoid the conceptual paradox of having the Any type being a supertype of for example a collection of Any types. The built-in basic type constructors we include are collections, sets, bags, sequences, etc. In fact we adopt that same set of operators as OCL [32] with the addition of structures and unions.

• **Data Types.** Data types refer to all basic types both built-in ones and constructed ones resulting from the application of built-in type constructors with the exceptions of component and service interface references. The idea behind this is that data types should not have associated behaviour, in order to simplify the definition of type relationships.

**4.2.7. Summary**

In this section we have identified the kinds of types that our component type model contains and presented the basic rational for our design choices. In some respects this rational will be further reinforced in the following sections. As a final point we
should say that the attachment of descriptions of behaviour to the types of our model and
the accommodation of semantic level interoperability information is the first argument
towards the characterisation of our approach as semantically enhanced. Additional
arguments to support our characterisation will be presented in the discussion of type
relations of our component type model.

4.3. Type naming

The names of types in both service trading and our semantically enhanced
component trading are very important both for the querying process and type
management. Regarding the querying, the whole process relies on identifying type.
Consumers of services or components request offers by naming their expected type. As a
result, traders only trade types they know, i.e. they are aware of their name. Regarding
type management, all operations like addition and deletion of types, introduction of type
relationships and the expression of relationships between types, name the types on which
their operations apply. So, the definition of a type naming scheme is a central part to the
definition of our component type model.

The main requirement for our type naming scheme is to be flexible and scalable
enough to support the naming of a large number of types. Schemes like universal
resource names (URNs) and universally unique identifiers (UUIDs) guarantee universally
unique type names and have been designed with scalability in mind. However, these
schemes are not really appropriate for our type model. First, using a universal naming
scheme is an overkill in our case, because the number of types that need to be known by
all traders is relatively small. Most of the types a trader will have to deal with are either
limited within a small number of application domains or even locally used. Second,
universal naming scheme do not use names that are meaningful for humans. In our case,
this is a problem. Meaningful type names allow for implicit information about the
intensions of the developer that introduces a type. Even though we recognise that this
kind of implicit information can be the source of a lot of confusion we still believe that it
is important for type names to be meaningful for humans. Therefore, in our case it seems
more appropriate to make type management more manageable through a naming scheme
that controls the scope of type names, instead of a universal one.
Hierarchical naming schemes like the one proposed by Indulska et al. for service trading in [1] are more appropriate for our component type model. The scheme is based on the observation that types can be classified into three categories:

- **Universal types.** Every type managers know these types. They are immutable and are defined by standardisation bodies. Basic types usually belong to this category.

- **Conformant types.** These are types that are standardised within a particular domain and are known to all type managers of the domain. These types are usually widely used within the particular domain to the extent that their standardisation became a necessity. Only appropriate authorities are allowed to assign names to conformant types.

- **Private types.** These types are defined by users and are only known to a particular type manager.

Accordingly, the hierarchical naming scheme defines a separate name space for each category of types, while the private name space can be further subdivided into a number of subspaces following a hierarchical structure.

A hierarchical naming scheme like the one described above mainly addresses the issues of type names from a type management point of view. The large number of type names though also poses challenges to the querying process. For this reason, it is important to control not only the scope of unique type names, but also the scope of searching for types. This is usually achieved with the introduction of contexts, which divide types into groups and restrict queries to only types of a particular context. In the simplest case contexts map directly to naming subspaces, but quite often they introduce further division of types still though in a hierarchical structure. Bearman and Raymond [33] present a more flexible approach, where users can define their own contexts for querying in terms of predicates expressed over attributes. Consequently, for the naming scheme of our component type model we follow a hierarchical approach similar to the one described above, which includes also the option of defining querying contexts.

### 4.3.1. Definition of our type naming scheme

We consider component types as private, i.e. only known to the trader they were introduced. Each trader defines a single private namespace for component types. This
namespace may be divided into a number of contexts, which are only used to restrict the search space during querying. When a query specifies a particular context, then only types defined within that context are considered. In general, the trader does not associate any semantic information to any of the contexts. During the introduction of new component types the trader considers the whole of its private namespace and guarantees that each component type name is unique. This means that the trader will consider potential component type relations between all component types even though they might belong to different contexts. However, any type relations transcending context boundaries will not be considered during context dependent querying.

We consider two kinds of service types: conformant and private ones. As we mentioned above, conformant ones are standardised within a particular application domain. Since in our type model we consider services as cohesive pieces of functionality, it seems reasonable to allow the standardisation of widely used services within a particular application domain. In order to avoid any name conflicts between private service types or ones standardised within different domains we require that the names of conformant service types to be qualified by the name of the application domain in which they are standardised. Although, conformant service types can be defined on their own, private service types can only be defined within a component type specification. Private service type names must be unique within the trader namespace they are introduced. So, in order to avoid any name conflicts between service types defined within different component type specification, we qualify the name of every private service type with the name of the respective component type.

Action types can only be defined as part of service type specifications. This means that their classification as private or conformant depends on the classification of the respective service type. The names of action types have to be unique within each trader's namespace. Accordingly, in order to avoid any name conflicts between action types defined within different service types, we require that their names be qualified by the name of the respective service type.

As we mentioned in section 4.2, we consider three kinds of property types: common, domain specific and local, referring to universal, conformant and private types respectively. The name of each property type must be unique for each trader's
namespace. Consequently, the names of common property types are standardised across all traders. The standardisation process guarantees the uniqueness of their names. The names of domain specific property types are standardised within a particular application domain and their names are qualified with the name of the respective domain. Finally, private ones are only defined as part of service or component type specifications and their names are qualified with the name of the respective service or component type.

Interface types are defined either explicitly or implicitly. In the latter case the interface type is defined through the specification of the respective type. These interface types are only introduced to our type model to assist in the description of relations between types. The trader automatically defines these types during the introduction of a new type, and uses them during the identification of relations between types. Their names are constructed by appending _I and _BI to the name of the respective type, for interfaces and behavioural interfaces respectively. The explicit definition is only available for behavioural service or component interface types. These types can be defined independently of component and service types. They can then be used in a number of component and service type definitions, allow the definition of types that support the same functional interface but differ on the property types associated with the interface. Behavioural component interface types are considered private and their name has to be unique within the namespace of the trader they were introduced. On the other hand, behavioural service interface types are either considered private or conformant. Conformant behavioural service interface types are standardised within a particular application domain and their name has to be qualified with the name of the respective domain. Private behavioural service interface types can only be defined as part of behavioural component interface types and their name has to be qualified with the name of the respective behavioural component interface type. Note that this view is inline with the view we take on private and conformant service types.

We consider all basic types as universal. This means that their names are standardised across all traders. This allows them to form the basis on which understanding of types between traders is achieved. We should note here that similarly basic type constructors are also universal.
4.3.2. Trader federation and type naming

Before we close the discussion on the type naming scheme we should discuss how name conflicts are resolved during trader federation. In federations with a shared type manager that maintains the type namespace, there is no issue of type name conflicts. On the other hand, in federations where multiple type managers are involved, name conflicts between private types may arise. In this case we resolve any such conflicts by qualifying all private type names with the name of the trader in which they were introduced. This, of course, will only work if each trader has a unique name within each federation. We should point out that guaranteeing unique trader names in federations does not really require a universal trader naming scheme. Such a scheme will be an overkill, since we expect each trader to only be in federation with a relatively small number of other traders. Moreover, the requirement for unique trader names within a federation does prevent traders of having different names in different federations.

4.3.3. Summary

We use a hierarchical type naming scheme in which each trader defines a separate name space for types. All type names within a trader’s namespace are guaranteed to be unique. We achieve this with the use of type name qualifications. Each trader’s namespace can be further divided into subspaces called contexts in order to control the scope of trader queries. The scope of each type name depends on its classification as private, conformant or universal. The names of universal types have a global scope and are guaranteed to be unique across all trader namespaces. The scope of the names of conformant types is restricted to the union of the namespaces of the traders that are aware of the application domain in which they are standardised. These type names are guaranteed to be unique within the particular application domain. While, the scope of the names of private types is restricted to the namespace of the trader in which they were introduced. These type names are guaranteed to be unique either within the trader namespace in which they were introduce in the case of component types and behaviour component interface types or within the container type, component or service type, or behavioural component or interface type, in which they were defined. Finally, interface types, behavioural or plain component or service interface types, may also be defined.
implicitly. In this case their name is automatically constructed from the name of the respective component or service type with the same scope.

4.4. Type description

In this section we present how each kind of type of our component type model is described in detail. This detailed presentation is a prerequisite for the definition of the type relationships of our model. Before we bring individual attention to each kind of types we start with the details of the model of behaviour for our types.

4.4.1. Describing models of type behaviour in UML/OCL

As we mentioned in section 4.2 components are just sets of services provided in context. Therefore, the focus in our model of behaviour for types is the service. Each service specifies its behaviour in a model-based manner. This involves first the definition of model types and model variables that capture the externally observed behaviour of the service. Then, the service behaviour itself can be specified through the use of invariants, history constraints, and the specification of the behaviour of its actions using pre- and post-conditions and model programs. Moreover, in section 3.3.1 we identified UML and its associated formal language OCL as a promising basis for the specification of type behaviour. Subsequently, we need to show how the combination of UML and OCL can be used to describe the various elements of service behavioural specifications.

In UML/OCL specifications there are two parts, a graphical part consisting of a UML class diagram and a textual part consisting of OCL annotations on the class diagram. In our descriptions of service type behaviour we maintain this separation between the class diagram and the OCL annotations, but our class diagrams are described textually in XML using XMI [34]. We consider the classes of the class diagram to represent user defined model types, one of which represents the model type of the service under specification. The attributes and the associations of the class representing the service model type define the model variables of the service behavioural specification, whereas the methods of this class represent the actions of the service.

In this context, the invariant of a service type is an OCL expression that describes a predicate over the attributes and associations of the service under specification. OCL
already supports the expression of invariants through the definition of the \texttt{<<inv>>}
stereotype. The history constraint of a service type is an OCL expression that describes a
predicate over subsequent states of the service. OCL currently does not provide support
for history constraints, but we can introduce a new stereotype, namely \texttt{<<hist>>}, for
this purpose. To represent the two subsequent states of the history constraint we can use
the names of the type variables with the \texttt{@pre} suffix, already provided by OCL, to refer
to instead of the immediately preceding state to any state before the current one, and as is
to refer to instead of the current state to any state following the current one. Since all
OCL expressions require the specification of a context on which the expression applies,
we consider the service type model class as the context of invariants and history
constraints.

Likewise, for the specification of the behaviour of the service's actions we can
use the already provided OCL stereotypes \texttt{<<pre>>} and \texttt{<<post>>} for pre- and post­
conditions respectively. The pre-conditions are predicates over the attributes and
associations of the service, model variables, and the input parameters of the action. The
post-conditions are predicates relating the state of the service before the execution of the
action, pre-state, to the state after its execution, the post-state. The pre-state is described
in terms of the model variables of the service with addition of the \texttt{@pre} suffix and the
parameters of the action. The post-state is described in terms of the model variables of the
service, the result of the action, if the action normal termination type is not void, and any
exceptional terminations raising exceptions. In fact, we can separate the normal and the
exceptional terminations by introducing alternative specifications as suggested by
Leavens and Baker in [35] and is supported by JML [36]. Each alternative specification
has each own pre- and post-condition. These alternative behaviours can be merged
together through the disjunction of the pre-conditions and the conjunction of pre­
condition implies post-condition predicates. Further improvements along the lines
suggested in [35] can be introduced through the definition of new stereotypes. The
context of the pre- and post-conditions is the action, whose behaviour they specify.

The introduction of model programs to UML/OCL specifications of behaviour is
a bit more challenging. OCL already includes constructs like conditionals (\texttt{if ...}
\texttt{endif}) and collection iterators (\texttt{iterate(...)}). The former can be used to express
conditionals in model programs and the latter to simulate loops. Although, OCL provides these constructs they are not widely used. Moreover, the implications of their use in the specification of model programs require further investigation. In this investigation the experiences and insights provided by JML can be very useful.

In the same way to the description of service behaviour, component type behaviour is also specified using UML/OCL. In the case of component types the class diagram of their behavioural specification combines all the classes diagrams of the respective service type class diagrams. Naming conflicts in the diagram can be resolved with the qualification of class names with the respective service type name. Equality between classes of the different diagrams can be established through the use of the component constraints. The component constraints express predicates that include elements, attributes, associations or operations, of classes belonging to classes of at least two separate service type model class diagrams.

In the discussion so far we have referred a number of times to model types without specifying what they are. In general, we take the view that model types could be any type that is acceptable as part of a valid OCL expression. The classes of the service type class diagram introduce, in fact, model types. However, in order for the operations of these classes to be usable in OCL expression they have to be side-effect-free. This means that they have to be just query operations. Note that this is inline with JML that requires any model types to be pure. In JML pure types must include methods that are pure, meaning they are side-effect-free.

In addition to these types, we also include the MODEL_VARIABLE type, which is a supertype of all model types, as a placeholder for required service type behavioural specifications (see also section 4.2). The specifier of a required service type creates a model for the behaviour of the service, but in this model instead of specifying the type of the model variables it uses the MODEL_VARIABLE type instead. Each model variable of MODEL_VARIABLE type introduces a placeholder that can be used in the predicates that specify the service type's invariant, history constraint and the pre- and post-conditions of its action types. During the matching process between provided and required services a unification (instantiation) of the each variable of MODEL_VARIABLE type of the required service to a corresponding state variable of
the provided service type is attempted. If the unification succeeds and the required logical relationships between the predicates of the required and provided services hold then the required and provided service types match. Finally, note that the component type constraint can instantiate certain model variables of MODEL_VARIABLE type to specific model types.

This completes the discussion on the detailed description of type behaviour. We can now go back to the detailed description of the various types, which will show where these behavioural specifications are placed. However, before we do this, we should point out that as it should be clear from the discussion in the previous section, names are an important part of all type descriptions. These names must conform to the type naming scheme introduced in section 4.3.

4.4.2. Component type description

Based on the discussion in section 4.2 component type descriptions besides the name of the type should also describe a set of services types, which includes both provided and required ones, a set of associated property types and any component type constraints. We allow two alternative ways of describing a component type:

- Verbose component type descriptions where all the elements of the component type are described in full detail.

- Compact component type descriptions where the component type is described in reference to a particular behavioural component interface.

More specifically, in the verbose way there are two sets of service type specifications: a provided and a required one, described using the keywords "provides" and "requires" respectively. The specification of both provided and required service types can take two forms:

- Named service types specifications use the keyword "named service". These specifications are only available for conformant service types that can be specified by just providing their fully qualified names. Remember that the fully qualified name of a conformant service type includes the name of the application domain within which the type has been standardised. For a named
service type specification to be valid its application domain has to be known to the trader at the time of the component type introduction.

- Detailed service types specifications use the keyword “described service”. These specifications can be used for private service type and require the full description of the service type as part of the component type specification.

The constraints of the component type are divided into two parts:

- **Service constraints.** These are specified using the keyword “service constraints”. They allow the expression of predicates that relate the behavioural models of the service types included in the component type specification (see discussion on the behavioural model in section 4.2 for more details).

- **Protocol constraints.** These are specified using the keyword “protocol constraints”. They refer to protocol interoperability and describe any constraints in the order of usage of the actions of the services that comprise the component.

While the above paragraphs apply only to verbose component type descriptions the description of component property types is applicable to both kinds of component type descriptions. In general, the property types included in component type descriptions provide placeholders for the description of overall characteristics that instances of the type have. These characteristics although they do not relate to any particular operational element of the component type, are still necessary to consider during the selection process and can seriously affect the suitability of component offers. For example, they could include things like the developer of the component, the compliance to particular standardised component model, the version, the creation date, the component storage footprint, memory needs, pricing information, etc. The specific values of these properties can be used to constrain and order the result set of a component offer request. For example, we might request only component offers by a specific author, or components offers on their 2nd or later version, or require that the offers be ordered according to footprint, or price, etc. Hence, component properties do not really describe characteristics that affect the operation of the services of the component per se such properties like QoS
related ones are only described as part of the particular service types. This is also the reason why we consider component property types to be atomic.

We identify two kinds of component properties:

- **Required properties.** Every component type must include these properties. We believe that a small set of essential properties should be standardised and required by all component types. The exact properties included in this set are in general up to the general community to decide through a standardisation process, but we can see the name of the developer, the release date and the version as some of them. Their types are universal and are described using the keyword “common property”.

- **Non-required properties.** These properties are up to the developer to include and define. Their types are either conformant or private and are described using the keyword “property”.

<table>
<thead>
<tr>
<th>Component Type $C := (N, S_{prov}, S_{req}, C_{CBM}, C_{prot}, CP, NP, DP)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N := $ name</td>
</tr>
<tr>
<td>$S_{prov} := $ a non-empty set of provided service types</td>
</tr>
<tr>
<td>$S_{req} := $ a potentially empty set of required service types</td>
</tr>
<tr>
<td>$C_{CBM} := $ a potentially empty set of constraints in reference to the component behavioural model CBM</td>
</tr>
<tr>
<td>$C_{prot} := $ a potentially empty set of protocol constraints</td>
</tr>
<tr>
<td>$CP := $ a non-empty set of required property types (common)</td>
</tr>
<tr>
<td>$NP := $ a potentially empty set of named property types</td>
</tr>
<tr>
<td>$DP := $ a potentially empty set of described property types</td>
</tr>
</tbody>
</table>

**Definition 4.1. Verbose Component Type Description.**

The specification of property types as with the specification of service types can take two forms: named and detailed, using the keywords “named property” and “described property” respectively. The former is only available for conformant property types that can be specified by just providing their fully qualified name. Note that required property types are also named and this is indicated by the use of the keyword “common”. The latter can be used for private property types and require the full description of the type itself. Furthermore, all property types are considered compulsory,
meaning that any instance of the type must have them, unless their specification is pre­
pended with the keyword “optional”. We also allow the specification of dynamic
properties, which require instances of the type to provide a reference to the code that
dynamically evaluates the value of the property. These properties are described by using
the keyword “dynamic property”, instead of just “property”. So, based on the
above discussion Definition 4.1 rigorously defines a verbose component type description.

Compact component type descriptions define the name of the type, the named
component behavioural interface and the component property types. Additionally, they
describe a set of service property types and their associated service types. The description
of the service property types is very similar to the description of component property
types discussed above. The only differences are: first, there are no common service
property types, second service property type descriptions are appended with the keywords
“associated to” followed by the list of service types they are associated to and third
service property types might be composite. Definition 4.2 rigorously defines a compact
component type description.

Component Type C := (N, CBI, CP, NP, DP, NSP^, DSP^)
N := name
CBI := component behavioural interface type
CP := a non-empty set of required component property types (common)
NP := a potentially empty set of named component property types
DP := a potentially empty set of described component property types
NSP^ := a potentially empty set of named service property types
associated to service types S
DSP^ := a potentially empty set of described service property types
associated to service types S
where S := services(CBI)

Definition 4.2. Compact Component Type Description.

4.4.3. Service type description

At this point, we should remind the reader that service type descriptions could
either be part of component type descriptions or exist on their own as service types
standardised within an application domain. In either case, the description of service types
has a number of similarities to the description of component types. For both kinds of
types there are two alternative ways of describing them: a verbose and a compact one.
The compact description of service types is in reference to a service behavioural interface
type. As component types include a set of service types, some constraints and a set of
properties types, service types include a set of action types, some constraints and a set of
property types. At the same time, the main differences are as follows:

1. Component property types may not be composite.
2. Action types in contrast to service types cannot be named.
3. The constraints part of service type descriptions is a bit more complicated than
   that of the component types.

\[
\text{Service Type } S := (N, A, \text{Inv}^{SBM}, \text{Hist}^{SBM}, C_{prot}, \text{NP}', \text{DP}')
\]

\[
N := \text{name}
\]

\[
A := \text{a non-empty set of action types}
\]

\[
\text{Inv}^{SBM} := \text{a potentially empty set of invariants in reference}
\]
\[
\text{to the service behavioural model SBM}
\]

\[
\text{Hist}^{SBM} := \text{a potentially empty set of history constraints in reference}
\]
\[
\text{to the service behavioural model SBM}
\]

\[
C_{prot} := \text{a potentially empty set of protocol constraints}
\]

\[
\text{NP}' := \text{a potentially empty set of named property types}
\]

\[
\text{DP}' := \text{a potentially empty set of described property types}
\]

**Definition 4.3. Verbose Service Type Description.**

\[
\text{Service Type } S := (N, \text{SBI}, \text{NP}', \text{DP}')
\]

\[
N := \text{name}
\]

\[
\text{SBI} := \text{a service behavioural interface}
\]

\[
\text{NP}' := \text{a potentially empty set of named property types}
\]

\[
\text{DP}' := \text{a potentially empty set of described property types}
\]

**Definition 4.4. Compact Service Type Description.**

More specifically, in the verbose way of describing services types the constraints
are divided into two parts: protocol ones and semantic ones, each referring to the
respective level of interoperability. According to our behavioural model of services, the
semantic constraints are expressed in terms of a model of the behaviour of the service. They are further divided into invariants and history constraints.

The description of atomic service property types is exactly the same to the description of component property types discussed in the previous section. The description of composite service property types is very similar to the description of atomic, but instead of using the keyword "property" we use the keywords "composite property". Definition 4.3 and Definition 4.4 rigorously define the verbose and compact service type description, respectively.

4.4.4. Action type description

| Action Type A := (N, NT, ET, P, PreSBM, PostSBM, MPSBM) |
| N := name |
| NT := normal termination type (return type) |
| ET := a potentially empty set of exceptional termination types |
| P := list \{(mode, name, type)\}, a potentially empty list of parameter types |
| PreSBM := a potentially empty pre-condition predicate in reference to the service behavioural model SBM |
| PostSBM := a potentially empty post-condition predicate in reference to the service behavioural model SBM |
| MPSBM := a potentially empty model program in reference to the service behavioural model SBM |

Definition 4.5. Action Type Description.

Action type descriptions include besides the action name, which is also the name of the type, a set of termination types, a list of parameter types and the action behavioural constraints. The set of termination types includes the return type of the action for normal termination and possibly the set of exception types for each exceptional termination. The parameter types form a list since their order is important. Each parameter type is a triplet of <mode, name, type>. The name of a parameter is a string and has to be unique within the list of parameter types. The type is the type of the parameter and can only be a basic type, a service or component behavioural interface reference, a data type, or a MODEL_VARIABLE type. The mode is either in, out or inout with exactly the
same semantics as in CORBA IDL [24], denoting parameter passing by value or by reference. Finally, according to our model of action behaviour the behavioural constraints of a method are divided into pre- and post-conditions and model programs. Definition 4.5 rigorously defines action type descriptions.

4.4.5. Property type description

<table>
<thead>
<tr>
<th>Atomic Property Type AT := (N, T)</th>
<th>Composite Property Type CT := (N, AP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N := name</td>
<td>N := name</td>
</tr>
<tr>
<td>T := type</td>
<td>AP := set of atomic property types</td>
</tr>
</tbody>
</table>

Definition 4.6. Atomic Property Type Description.

Definition 4.7. Composite Property Type Description.

At this point, we should remind the reader that property types can be described either as part of component or service type description, or on their own either as standardised types within a particular application domain, or as universally standardised common property types. The former case has been discussed in the previous sections here we only focus on the latter case. We should also remind the reader that there are two kinds of property types: atomic and composite ones. Atomic property type descriptions include a property name, which is also the name of the property type, and the type of the property value that can only be either a basic type or a data type. Composite property type descriptions include besides the name of the property, a set of atomic property types. Definition 4.6 and Definition 4.7 rigorously define atomic and composite property type descriptions.

4.4.6. Component and service behavioural interface type description

The description of component and service behavioural interface types is very similar to the verbose description of component and service types respectively with the removal of property types. Definition 4.8 and Definition 4.9 rigorously define the respective type descriptions.
Component Behavioural Interface Type CBI := \( (N, SBI_{prov}, SBI_{req}, C^{CBM}, C_{prot}) \)

- \( N := \) name
- \( SBI_{prov} := \) a non-empty set of provided service behavioural interface types
- \( SBI_{req} := \) a potentially empty set of required service behavioural interface types
- \( C^{CBM} := \) a potentially empty set of constraints in reference to the component behavioural model CBM
- \( C_{prot} := \) a potentially empty set of protocol constraints

**Definition 4.8. Component Behavioural Interface Type Description.**

Service Behavioural Interface Type SBI := \( (N, A, Inv^{SBM}, Hist^{SBM}, C_{prot}) \)

- \( N := \) name
- \( A := \) a non-empty set of action types
- \( Inv^{SBM} := \) a potentially empty set of invariants in reference to the service behavioural model SBM
- \( Hist^{SBM} := \) a potentially empty set of history constraints in reference to the service behavioural model SBM
- \( C_{prot} := \) a potentially empty set of protocol constraints

**Definition 4.9. Service Behavioural Interface Type Description.**

### 4.4.7. Component, service and action interface type description

Component Interface Type CI := \( (N, SI_{prov}, SI_{req}) \)

- \( N := \) name
- \( SI_{prov} := \) a non-empty set of provided service interface types
- \( SI_{req} := \) a potentially empty set of required service interface types

**Definition 4.10. Component Interface Type Description.**

Service Interface Type SI := \( (N, AI) \)

- \( N := \) name
- \( AI := \) a non-empty set of action interface types

**Definition 4.11. Service Interface Type Description.**

143
Action Interface Type (action signature) \( AI := (N, NT, ET, P) \)

- \( N := \text{name} \)
- \( NT := \text{normal termination type (return type)} \)
- \( ET := \text{a potentially empty set of exceptional termination types} \)
- \( P := \text{list}\{\text{mode, name, type}\}, \text{a potentially empty list of parameter types} \)

**Definition 4.12. Action Interface Type Description.**

The description of component, service and action interface types is very similar to the description of component and service behavioural interface types, and action types with the removal of all the behavioural constraints parts. Definition 4.10, Definition 4.11 and Definition 4.12 rigorously define component, service and action interface types respectively.

### 4.4.8. Basic types, constructed types and data types description

Basic types are described by naming the type. Constructed types are described by naming the type constructor and the basic type on which the constructor is applied. Data types are described by naming the type and a set of <name, type> pair for each of the constituent basic types. The definitions of the descriptions of all these types should be quite straightforward and for this reason we do not provide them.

### 4.4.9. Summary

In this section we specified how the various types of our type model are described. We started with a discussion on how the model of behaviour for service and component types can be described in UML/OCL, followed by a specification of the description of each individual kind of type. Additionally, for all types we provide a rigorous definition of their description, which we will be further utilised in the definition of the type relationships of the type model. Furthermore, for component and service types we offer both a verbose and a compact way of description, allowing the description of these kinds of types in reference to a corresponding interface type. Finally, we define a number of keywords as part of the description of the various kinds of types. These keywords form a preliminary core for the definition of a component description language based on the type model.
4.5. Type relationships

In the previous sections we focused on the types of our component type model, and we discussed the issues pertaining to their naming and description. In this section the focus shifts to the relationships between these types. In general, there is a plethora of potential relationships between types that one could define. However, in this thesis we only focus on relationships that can be utilised during trading. We start the identification of such kind of relationships by reminding the readers of the way in which type relationships are currently utilised in service trading.

As we saw in section 3.2, in service trading service offers have a service type and the trader matches service requests to service offers on the basis of their type. During the trading process, relationships between service types can be used to expand the search for matching offers. In this case, the trader may consider as matching offers not only those that have the requested type, but also those that have types related to the requested one. For these additional offers, though, to be useful to the requestor, they must satisfy its requirements, or in other words be compatible/conformant to the requested service type. Consequently, all type relationships used by the trader must provide certain guarantees of compatibility/conformance for their respective offers. These guarantees are usually expressed by the "no surprise rule". The rule states that if we replace an offer of a particular type with an offer of any compatible/conformant type, there should be no surprises, both in terms of errors and unexpected results. Hence, in service trading the key type relationship is service type compatibility/conformance expressed by the "no surprise rule".

At this point we should draw attention to some interesting points regarding type compatibility. According to Brookes and Indulska [39] the key issue in type compatibility is substitutability, which the deceptively simple "no surprise rule" also indicates. The deceptive simplicity of the rule is the result of the fact that it does not specify the kinds of errors and unexpected results that it refers to. In any system there are a variety of different types of unexpected behaviour that one can consider. Considering a different scope for unexpected behaviour leads to a different definition of type compatibility. As we mentioned in section 3.2, service trading usually ignores this variety and only considers a single definition. This approach restricts unnecessarily the number of types
considered during the query expansion phase and as a result limits reuse. In contrast to service trading in our component type model we attempt to maximise reuse and flexibility by considering a variety of type compatibility definitions. As we saw in section 3.3, the way in which we are trying to achieve this is by expanding the narrow view of compatibility in service trading to cover besides signature-level interoperability, semantic-level too. Remember that because protocol level interoperability information can be considered as guidelines of correct usage, we decided to ignore this kind of interoperability issues in the type model. Incorporating additional interoperability levels in our view of compatibility further reinforces the need to introduce alternative definitions for conformance in our type model. We could, for example, consider a separate definition of compatibility for each interoperability level.

Before we delve into the details of the various conformance definitions it is necessary to set a framework within which these definitions will be placed in order to clarify the relationships between them. The issues of how type relationships are defined and related have been extensively studied in the context of typing in programming languages. As a result, it is within this area of work that we expect to find the required framework. More specifically, as we have adopted an object-based model of services and components, it makes sense to focus our search in the area of object-oriented typing. In the context of object-oriented languages, the discussion on type compatibility focuses on the role of types and the definition of subtyping. Note that as we have seen in section 3.2, in service trading conformance between services is quite often expressed in terms of interface subtyping, making the work on object-oriented typing particularly relevant.

4.5.1. A framework for type relation definition based on object-oriented typing

Within the area of object-oriented typing, Palsberg and Schwartzbach were the first to attempt to codify all the various notions of subtyping adopted by different languages into a unified framework [57]. The unified framework is based on the realisation that the role of types in object-oriented languages is to impose some discipline in the use of objects. This is achieved by restricting the objects on which methods can be called and the objects that can be passed as parameters to these methods. These
restrictions can be expressed as predicates that all acceptable objects must satisfy. These predicates can be in turn viewed as definitions of types. As a result, the acceptable objects are only those that adhere to a specific type. Note that this predicate-based view of types means that in general an object will have many types.

In general, the predicates that define types can take almost any kind of form, each considering different aspects of the objects. Since, in object-oriented languages classes are used to describe the implementation of objects, these predicates consider different aspects of the various class features, i.e. attributes and methods. Accordingly, the whole class can be used as a type, where the predicate requires that the object is an instance of a class or any of its subclasses. In this case, it is the choice of subclassing mechanism that specifies the exact predicate. Palsberg and Schwartzbach recognised that the various approaches encountered in the literature can be arranged into a sequence according to their expressiveness, i.e. the number of class features they consider. This sequence starts with the case of arbitrary subclassing where the predicate is trivial, followed by name compatibility, where the subclasses are required to share a number of methods with the same names. It continues to interfaces, where the subclasses are required to share a set of methods with the same name and the same types of arguments, followed by class and monotone subclasses, allowing only the addition of new methods or the redefinition of the method bodies of existing ones, and where the predicate requires that objects have a common set of methods with the same name and the same types of arguments. It goes on to behaviour, where in addition to sharing a set of methods with the same name and types of arguments, the behaviour of the methods, typically specified by pre- and post-conditions, is also part of the predicate. It ends with class and strictly monotone subclasses, where the predicate requires that objects have a common set of methods, which in addition to the name and the types of their arguments also have the same implementation.

According to the above arrangement of type definitions in terms of classes and subclassing based on their expressiveness, subtyping in object-oriented languages is also defined with respect to these type definitions. This is because subtyping is a partial order, such that if type $T_1$ is a subtype of type $T_2$, then any object of type $T_1$ is also an object of type $T_2$. This means that the subtypes must provide the same guarantees as their
supertypes, i.e. all subtypes must adhere to the predicate that defines the supertype and as a result are by definition compatible with their supertype. So, the relationships between definition of type and subtyping are as follows:

- When the type is class and subclasses, then subtyping becomes subclassing.
- When the type is name compatibility, then subtyping becomes more methods.
- When the type is interface, then subtyping becomes interface conformance.
- When the type is behaviour, then subtyping becomes behavioural subtyping.

An interesting observation about the above definitions is that they allow subtype objects to have additional features/methods as long as these features maintain the guarantees provided by the superclass.

From the discussion so far, it should be clear that there are a number of aspects of methods that can be considered in type definitions in object-oriented languages. These aspects determine also the exact nature of subtyping and are the following:

- Names
- Interfaces
- Behaviour
- Implementation

Moreover, these aspects are considered ordered in terms of expressiveness, with names being the least expressive and implementation the most expressive one. In this sense each more expressive aspect considers also all the less expressive ones. In other words a type predicate providing guarantees for method implementations provides also guarantees for method behaviour, interfaces and names. In all cases, the selected predicate for the definition of the type, which the subtype needs to adhere to as well, is also the condition that determines type compatibility.

Following an approach similar to the one followed by Medvidovic et al. in their type theory for software architectures [29], we also consider the above four aspects as the basis in specifying the dimensions of the framework within which our type relationships will be defined. However, since in the component type model, types definitions do not consider the implementation of the components, as component types abstract away from the implementation details, we will exclude the implementation aspect. As a result our framework will only have three dimensions each reflecting one of the names, interface
and behaviour aspects. Consequently our various notions of type compatibility/conformance will be expressed in terms of predicates along these three dimensions of the framework.

4.5.2. Refining the framework within the solution context

In the previous section we identified name, interfaces and behaviour as the three dimensions of our framework for the definition of type relationships. However, it is not clear how these dimensions relate to work on component retrieval in software reuse and the three levels of interoperability we examined in section 3.3, i.e. the context within which SECT type model is defined. In this section we refine the framework sketched above in order to accommodate the various elements of the SECT type model context.

We start by examining how the dimensions of the framework relate to the three interoperability levels. It should be clear that the interface dimension corresponds to signature level interoperability. At the same time, the behaviour dimension addresses all three interoperability levels. This is the case because as we mentioned above each of the three dimensions also incorporates all the dimensions below it, i.e. behaviour incorporates also interface, which in turn incorporates names. At the same time, behaviour in general, if it is described at an appropriate level of expressiveness, incorporates both protocol and semantic level interoperability. However, as we mentioned in section 3.3.2, we prefer to address each interoperability level separately. As a result, the three dimensions need to be rearranged to separate the different interoperability levels. Moreover, in section 3.3.2 we also decided to exclude protocol level interoperability from the considerations of our type model. Accordingly, we replace the behaviour dimension with a semantics dimension that captures only those aspects of behaviour pertaining to the semantic interoperability level. As a result, the semantics dimension no longer fully covers the other two dimensions, allowing us to also consider relationships between types in terms of their semantics even though there is no relationship between their interfaces. An example of such a kind of relationship is the ontological object semantic compatibility defined in [190]. According to the definition such a relationship holds between two object methods that perform exactly the same function, i.e. have the same behaviour, but operating on parameters that can be shown to
be semantically equivalent even though their type structure and name does not indicate so (e.g. defining date as day-month-year triplets or as the Unix format of seconds since January 1st, 1970). We should note that this kind of relationships is difficult to establish, without the use of annotations (ontology markup references) to the semantically equivalent types. The kind construct defined in [190] aims to capture such annotations.

As the separation of the semantics dimension from the other two, seems to increase the range of type relationships we can consider, this may also be the case if we separate the names and interfaces dimensions too. First, we should point out that as we discussed in section 3.2.1, names although necessary to enable interoperation between services can be in fact the source of misconceptions and misunderstandings. It is for this reason, that component retrieval approaches in software reuse have excluded method or operation names from the signatures they consider. This is particularly demonstrated in the case of signature matching of software components [167]. However, ignoring the names of the operations presents problems when we consider the "no surprise" rule, since there is no guarantee that the callers of an operation will know how to invoke it. These problems though can be easily resolved with the use of interface adapters [114] that can mask any interface differences. In fact, the construction of such adapters can easily be automated. So, in general it seems quite restrictive to consider both the structure and the names of type features within the interface dimension. Accordingly, we replace the interface dimension with a structure dimension, which defines relationships between types in terms of only the structure of their elements. As a result, the structure dimension no longer incorporates names, allowing us to take advantage of the work on signature matching [167]. As we now have to a certain degree separated the three dimensions of our framework, we need to keep in mind that type relationships that do not provide guarantees in terms of both structure and names, need to be handled with care in the trading process.

Having identified the three dimensions of our framework for type relationships and following an approach similar to the one described by Medvidovic et al. in their type theory for software architectures [29], we can view each dimension as defining a class of compatibility/conformance relationships. We are referring to a class of relationships because the dimensions only prescribe which aspects of the type features are considered
in defining a relationship predicate. A specific relationship predicate determines an instance of a particular class. We can depict these classes graphically as areas (circles) of the overall type relationships area (see Figure 4.2). There are two important observations regarding the figure. First, the three classes are not disjoint, since it is possible to define compatibility/conformance relationships that incorporate two particular dimensions or even all three of them. For example, we could define a type relationship that requires the same semantics, the same names and the same structure for the common features of related types. Second, relative sizes of the circles as well as the overlaps do not carry any information. This means that we are not making any claims regarding the number of particular type relationships that belong to each class. Furthermore, we can use the same figure to depict the various kinds of compatibility/conformance as specific areas of graph. For example, the class of syntactic notions of conformance, like the one supported by service trading (see the four rules of conformance in section 3.2), is depicted in Figure 4.3. The figure clearly demonstrates that this kind of notions of conformance do not necessarily incorporate semantic interoperability, as part of the marked area is outside the semantics circle. As we define the various classes of compatibility/conformance relationships in the following sections, we will also show where these are positioned on the graph.

4.5.3. Name-based compatibility/conformance

Type relationships in this class require that the related types comprise of "features" with names satisfying a particular predicate, and they cover the whole of the
Names circle in Figure 4.2. Therefore, these relationships cannot be defined between "atomic" types, i.e. types that are not defined in reference to other types, like basic types or atomic property types, as they have no "features" to refer to. Consequently, we define name-based compatibility/conformance for the following types:

- Data types, where the features are the types they comprise of.
- Composite property types, where the features are their atomic property types.
- Service interface types, where the features are their action interface types.
- Component interface types, where the features are their service interface types.
- Service behavioural interface types, where the features are their action types.
- Component behavioural interface types, where the features are their service behavioural interface types
- Service types, where the features are their property types, and either their action types or the action types of their service behavioural interface type.
- Component types, where the features are their property types, and either their service types or the service types of their component behavioural interface type or their service types.

Note that since all the features of the above types are types of our type model, they all have a name, which is what the predicate of the name-based relationships refers to. The most obvious such predicate is equality of corresponding feature names. However, as our type naming scheme guarantees the uniqueness of the fully qualified type names, this kind of relationships only make more sense if we consider the unqualified type names. As we have mentioned a number of times so far, in general name-based compatibility/conformance can be dangerous, unless the names are in some sense "standardised", i.e. not randomly select, or some care has been taken to ensure that they really convey certain meaning. We will see how an appropriately selected predicate achieves the latter later on (see the definition of domain name relationship in section 4.7). Even when no care is taken with respect to the meaning of the names, this kind of relationships is also useful as a compliment to other kinds.

From the possible name-based type relationships, here we only define the basic name compatibility between types. Moreover, instead of rigorously defining this
relationship for all applicable types, we provide a rigorous definition for the generic basic name compatibility as follows:

Type $T_1$ is "basic name compatible" to type $T_2$ if and only if

$\forall$ feature $f_i$ of $T_2$, $\exists$ feature $f_j$ of $T_1$ such that $\text{nameOf}(f_i) = \text{nameOf}(f_j)$

Definition 4.13. Generic basic name compatibility.

The particular basic name compatibility relationships can be defined from the generic definition, by replacing the type with a particular kind of type and the feature with the specific kind of feature according to the list provided above. Note that the generic definition allows type $T_1$ to have more features than type $T_2$.

4.5.4. Structure-based compatibility/conformance

Type relationships in this class require that the structures of the related types satisfy a particular predicate, and they cover the whole of the Structure circle in Figure 4.2. In order to determine what kind of relationships this class includes we need to first determine what is the structure for the various types of the component type model. So, in the case of basic types the structure is the type itself, in the case of constructed types their structure includes the type constructors and the type on which the constructor is applied to, while in the case of data types their structure is the set of types they comprise of. In the case of property types, for atomic ones their structure is their type, while for composite ones their structure is the set of atomic property types they comprise of. In the case of action interface and action types their structure is the action signature, which comprises of the mode and type of their parameters, the type of the normal termination, and the types of the exceptional terminations. In the case of service interface types and service behavioural interface types, their structure is the signatures of the list of action types or action interface types they include. In the case of service types, their structure is the structure of the respective service behavioural interface type, together with the structure of their properties. In the case of component interface types and component behavioural interface types, their structure is the structure of the set of provided and required service interface types and service behaviour interface types, respectively. Finally, in the case of component types their structure is the structure of the respective component behavioural interface type, together with the structure of their property types.
At the heart of this kind of type relationships is the work on signature matching [167, 191]. This work primarily focuses on function and module signatures, with the latter being defined in terms of the former. Function signatures correspond to abstract forms of our actions, in the sense that they view actions as tuples of types instead of separating between parameter, normal termination and exceptional termination types. More specifically, in this work every match between function signatures is generically defined as a relationship between the corresponding types of the tuples after the application of certain transformations to these tuples (see Definition 4.14). The various matches are classified into two categories: partial ones, where the relationship between the corresponding types is varied from equality, to type specialisation and type generalisation; and transformation ones, where the transformations between the corresponding types is varied from type variable and type constructor renaming, to un-currying of higher-order functions, and argument reordering. Besides, the basic matches the work also defines a number of composite ones, which combine two or more of the basic ones.

\[
\text{Two function types } f_1 \text{ and } f_2 \text{ match}
\]

\[
M(f_1, f_2) = \exists \text{ transformation pair } T = (T_1, T_2) \mid T_1(f_1) R T_2(f_2)
\]

**Definition 4.14. Generic Signature Match**

From all the different matches we mentioned above, we identify the following ones as particularly suited for the definition of structure-based compatibility/conformance relationships:

- **Exact match.** In the context of the component type model this match is interpreted to mean that there is a one-to-one correspondence between the structure elements of the actions, and that the corresponding structure elements are of the same type. In the case of the parameter types, we require that besides their type, their modes are also the same. Moreover, since the parameter types are organised into a list, the particular parameters are considered to have a specific position within this list. As a result, the correspondence of parameters is with respect to their position within the list. In contrast to parameters types, the exceptional termination types are not ordered. Furthermore, the requirement for a one-to-one correspondence
between structure elements means the two actions should have parameter lists of the same size. Type equality is determined on the basis of the fully qualified names of the types.

- **Reorder match.** This is very similar to the exact match. The only difference is that it does not consider the position of the parameter types in list as important. In other words, allows us to change the order of the parameters before we establish an exact match between the actions.

- **Type constructor renaming.** This match is also very similar to the exact match. The main difference is that instead of requiring the equality of corresponding types, it allows us to ignore the names of non-universal types provided that they have the an equivalent structure. This requires the definition of some notion of structure equivalence for the different kinds of types. We come back to this issue later on.

- **Contra-/co-variance.** This is a partial match that captures the covariance of results and contra-variance of arguments rule for subtypes in object-oriented languages. In the context of the component type model, this rule requires that an action type A conforms to an action type B, if and only if A’s normal and exceptional termination types conform to B’s corresponding normal and exceptional termination types (covariance), and B’s parameter types conform to A’s corresponding parameter types (contra-variance). This would have been adequate in the case where parameter types do not have modes. However, in our case we need to also take this modes into consideration as follows:
  
  - in parameters follow the contra-variance rule, i.e. B’s in parameters must conform to A’s corresponding in parameters.
  - out parameters follow the co-variance rule, i.e. A’s out parameters must conform to B’s corresponding out parameters.
  - inout parameters follow the invariance rule, i.e. A’s and B’s corresponding inout parameters must be equivalent, with respect to some notion of structure equivalence.

Besides the above basic matches, of particular interest are also the composite matches that combine the latter three. Regarding these composite matches Zaremski in
[191] makes a number of observations. First, applying the type constructor renaming and the reorder matches more than once does not really make any difference (see Theorem 2.4.4 in [191], p. 30). Second, the composition of these two matches is commutative, i.e. the order in which the two matches are applied is not important (see Theorem 2.4.2 in [191], p. 27). The same is not the case when all three matches are combined as the transformation matches (type constructor renaming and reorder matches) need to be applied first.

As we mentioned in the previous section, structure compatibility/conformance relationships in some cases require particular care when used in the trading process, since they do not necessarily guarantee that a conformant type will be directly usable. From the above discussion, it should be clear that such relationships are particularly those that involve reordering of parameters, renaming of types, or even, depending on the support of the underlying platform, contra-/co-variance matches. These relationships require some form of adaptation or wrapping of the conformant actions. This requirement introduces to the component type model the notion of composite relationships, i.e. ones that require some form of adaptation or wrapping of the interface of conformant types. It also raises the question of how are these adaptations or wrappings determined and carried out. In some cases this process can be automated, since the compatibility/conformance relationship itself determines how the process should be carried. A good example is the case of relationships involving the reordering of action parameters. In other cases, automation is not possible and custom adapters are required for the relationship to become usable. The distinction between these two cases introduces to the component type model the notion of runtime and development time type relationships respectively. We will come back to this issue when we present a trader architecture for SECT.

As we mentioned above, both type constructor renaming and contra-/co-variance for inout parameters require some notion of structure equivalence. In order to define such relationships we utilise Zaremski’s classification of the various signature matches as equivalences, partial orders or neither [191]. According to this classification, a match relation is an equivalence match if it is reflexive, symmetric and transitive, while it is a partial order match if it is reflexive, anti-symmetric and transitive. From the matches we have discussed above the exact match and the two transformation matches (type
constructor renaming and reorder matches) are equivalences, while we can show that the partial match (contra-/co-variance match) is a partial order. Furthermore, the composite match composing the two transformation matches is also an equivalence (see Theorem 2.4.5 in [191], p. 30), while we can show that the combination of the latter three matches is a partial order. As a result, we can use for the definition of structure equivalence either type equality or any of the equivalence matches. We should also note that the partial order matches could be used in the definition of subtyping relationships. We will come back to this issue later on.

The discussion so far has been focused on structure-based compatibility/conformance of action and action interface types. Let us now examine how to extend these relationships to the other types of the component type model. In the case of service interface types and service behavioural interface types, the extension is straightforward. It requires that the corresponding action interface types or action types respectively are structure-based conformant according to one of the above matches. Note that usually the same match is used for the compatibility/conformance of all action interface types or action types. In the case of component interface types and component behavioural interface types the extension is also relatively straightforward. In this case, we need to determine a separate correspondence of provided and required service interface types and service behavioural interface types respectively. Moreover, extending the contra-/co-variance rule, we require contra-variance of required and covariance of provided service interface types and service behaviour interface types. Note that this is similar to the matching rules for components with multiple interfaces defined in [192]. With respect to atomic property types, we require that the types of the properties are structure equivalent in the sense described above, while in the case of composite property types we require structure conformance of the corresponding atomic types. Note that in the context of property types only type equality, exact match and type constructor renaming match apply. This means that the only flexibility we allow is the renaming of non-universal types. We can now go on to extend structure-based compatibility/conformance to service and component types. In both cases we require structure conformance of the corresponding property types and the corresponding service and component behavioural interface types respectively.
4.5.5. Syntactic compatibility/conformance

Type relationships in this class combine name- and structure-based compatibility/conformance relationships. As a result, they require that the related types comprise of "features" with names satisfying a particular predicate that are also structure conformant. The area covered by this class is depicted in Figure 4.3. Since they require name-base compatibility/conformance, these relationships are only defined over the kind of types identified in section 4.5.3. From the plethora of possible relationships we will only focus on the kind of relationships that are not composite, i.e. are directly usable (see section 4.5.4). These are relationships in which the structure compatibility/conformance is determined only in terms of type equality or exact match, while the name predicate is just equality of the names of the corresponding features.

More specifically, in the case of service interface types and service behavioural interface type, we requires that the corresponding action interface types and action types respectively, have the same name and are exact match structure conformant. It is easy to see that this definition is in fact the definition of interface conformance used in service trading. In the case of service types, in addition to the syntactic conformance of their respective service behavioural interface types, we also require that the corresponding property types have the same name and either structure conformant type for atomic properties, or syntactic conformant composite property types (i.e. with corresponding atomic properties with the same name and structure conformant type). In the case of component interface types, component behavioural interface types and component types, the definition of syntactic compatibility/conformance is similar, but with reference to the corresponding service interface types, service behavioural interface types and service types respectively. Note though that in all case the structure conformance follows contra-variance of required and covariance of provided service-related types. This is in fact a natural extension of the interface conformance of service trading to multi-interface component conformance.

4.5.6. Semantics-based compatibility/conformance

Type relationships of this class only apply to types that include descriptions of behaviour and they define a particular predicate that these descriptions should satisfy. In
general, this class covers the whole of the semantics area in Figure 4.2. However, it is not as straightforward as it might seem at first. The reason is that the behaviour of types, like component types, component behavioural interface types, service types and service behavioural interface types is described with respect to a behavioural model (see section 4.2.5). This requires that in order to be able to relate their behaviour specifications in the relationship predicate, we need to be able to either translate the two models into a common model, or translate one of them into the other. In order to deal with this most relationships between behavioural specifications utilise either abstraction or simulation functions that aim to achieve this goal (see [193] for example), in the general case abstraction relations are needed instead. Finding this kind of functions is not easy unless the behavioural models of the specifications are quite similar in form. Furthermore, particularly in the case of action types, the behavioural specifications also refer to structural elements of the type itself, e.g. the parameter types and the termination types. As a result, a structure-based relationship between the types usually simplifies matters further. This is why work on specification matching [166] also requires signature matching [167] between related types. In conclusion, despite the fact that this class of relationships covers the whole of the semantics circle, the relationships we will consider only cover part of this area in practical terms.

At the heart of our approach is the work on specification matching by Cheng [194, 195], and Zaremski and Wing [166, 191]. This work is primarily built around the notion of specification matching of function behaviour specifications. The behaviour specifications are in terms of action pre-/post-conditions and the matches are expressed as a predicate over them. Since our action types behaviour is also described in terms of pre-/post-conditions these matches can be directly applied to define semantics-based relationships between them. However, as our action types may also define a model program, these relationships have to be extended to include as part of their predicate these model programs too. In Table 1 we list all the different specification matches encountered in the literature [166, 191, 195]. Each definition specifies the predicate that the pre- and post-conditions of a query function specification (Q_pre and Q_post) and a selected function specification (S_pre and S_post) must satisfy. This work also explores the relation between various matches in terms of logical implication or equivalence, which
are then used to organise the various matches into a lattices. However, we should note that although there is a set of function specification matches considered in all of [166, 191, 195], each of them also introduces some of its own. Moreover, each of them also provides a slightly different lattice, showing only the matches they consider. Therefore, before we can continue the discussion we need to consolidate all the matches of Table 1 in lattice like structure.

Table 1: Function Specification Matches

<table>
<thead>
<tr>
<th>Match Name</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact pre/post</td>
<td>$(Q_{pre} \leftrightarrow S_{pre}) \land (S_{post} \leftrightarrow Q_{post})$</td>
</tr>
<tr>
<td>Exact pred.</td>
<td>$(S_{pre} \Rightarrow S_{post}) \leftrightarrow (Q_{pre} \Rightarrow Q_{post})$</td>
</tr>
<tr>
<td>Exact pred. 2</td>
<td>$(S_{pre} \land S_{post}) \leftrightarrow (Q_{pre} \land Q_{post})$</td>
</tr>
<tr>
<td>Plug-in</td>
<td>$(Q_{pre} \Rightarrow S_{pre}) \land (S_{post} \Rightarrow Q_{post})$</td>
</tr>
<tr>
<td>Guarded plug-in</td>
<td>$(Q_{pre} \Rightarrow S_{pre}) \land ((S_{pre} \land S_{post}) \Rightarrow Q_{post})$</td>
</tr>
<tr>
<td>Plug-in post</td>
<td>$S_{post} \Rightarrow Q_{post}$</td>
</tr>
<tr>
<td>Relaxed plug-in</td>
<td>$(Q_{pre} \Rightarrow S_{pre}) \land ((Q_{pre} \land S_{post}) \Rightarrow Q_{post})$</td>
</tr>
<tr>
<td>Guarded gen. pred.</td>
<td>$(Q_{pre} \Rightarrow S_{pre}) \land ((S_{pre} \Rightarrow S_{post}) \Rightarrow (Q_{pre} \Rightarrow Q_{post}))$</td>
</tr>
<tr>
<td>Guarded post</td>
<td>$(S_{pre} \land S_{post}) \Rightarrow Q_{post}$</td>
</tr>
<tr>
<td>Weak post</td>
<td>$S_{pre} \Rightarrow (S_{post} \Rightarrow Q_{post})$</td>
</tr>
<tr>
<td>Generalised</td>
<td>$(S_{pre} \Rightarrow S_{post}) \Rightarrow (Q_{pre} \Rightarrow Q_{post})$</td>
</tr>
<tr>
<td>Specialised</td>
<td>$(S_{pre} \Rightarrow S_{post}) \leftarrow (Q_{pre} \Rightarrow Q_{post})$</td>
</tr>
</tbody>
</table>

The resulting structure is provided in Figure 4.4. Most of the relationships in this figure are directly taken from the lattices provided in [166, 191, 195]. These relationships are depicted using black arrows. In addition to these the equivalence relation between Guarded post and Weak post, although does not appear in any of these lattices is mentioned by Zaremski in [191] and for this reason is depicted by a pink arrow. The relations between Exact pred 2 and Guarded post, and Guarded gen. pred. and Generalised are our own provision and are therefore depicted using red arrows. Our proof of these relations is relatively straightforward and is based on the properties of the logical operators.
The variety of function specification matches reflects the diverse needs that they address. For example, in [191] the following potential uses are identified: (a) retrieval of functions from a library, either for reuse of their code, for statistical analysis of the library or retrieval-based browsing of the library, (b) indexing of the library, and (c) substitution, especially in the definition of behavioural subtypes. Therefore, keeping in mind that these matches will form the basis for semantics-based relationships between types for SECT, we need to select the most appropriate of them. These relationships are used in SECT to retrieve components that can as far as possible be seamlessly used in the required context. Having a similar motivation Chen and Cheng in [195], they define reusability to formally capture this notion of seamless use in a particular context (see Definition 4.15). Note that the correctness formula for program $A$ can be informally interpreted as follows: "$\{p\}A\{q\} = \text{truth of program } A \text{ begun with } p \text{ satisfied, will terminate with } q \text{ satisfied}" [195]. Based on this definition, they continue to define the notion of reuse ensuring match (see Definition 4.16). Furthermore, they also show that the reuse ensuring matches with logical implication form a lattice and that the Relaxed plug-in match is the most general reuse-ensuring match. Note that in Figure 4.4, all reuse-ensuring matches are depicted with a turquoise bounding box. It should be clear
that the set of reuse-ensuring matches provides fairly strong guarantees in terms of component reusability and as such are at the core of our semantics-based compatibility/conformance relationships.

Given a query specification $Q : (Q_{pre}, Q_{post})$ a component $A$ is reusable for implementing $Q$, if $\{Q_{pre}\}^A \{Q_{post}\}$ holds, where $\{p\}^A\{q\}$ is the correctness formula for program $A$.

Definition 4.15. Component Reusability.

In addition to the reuse ensuring matches we further increase the range of our semantics-based relationships by also considering the Guarded post and Weak post matches. These two matches although not reuse-ensuring, they can be usable in contexts where we can guarantee the truth of their precondition. The requirement for such guarantees means that in contrast to the reuse ensuring ones, these matches are not directly usable. As a result, the relationships that use them are composite in the sense described in section 4.5.4. Guaranteeing that these relationships are runtime ones requires that we provide a façade [114] that would only call these conformant components, when their precondition is satisfied, calling some other component in the cases when the precondition is not satisfied. From this discussion, it should be clear that this idea of using composite relationships to adjust the basic components can be generalised to incorporate in addition to one-to-one compatibility/conformance relationships, one-to-many ones too. In conclusion, Figure 4.5 depicts the lattice of our selected function specification matches.

Having identified the function specification matches that we are going to use as a basis for the semantics-based relationships between action types, we now need to come to the issue of how their model programs may be related. In general, model programs can be related with some notion of program refinement, which builds upon the theory of code.
refinement. The idea behind code refinement is that abstract statements can be gradually refined into more specific statements until they are specific enough to be expressed in a particular programming language. A calculus has been devised, the refinement calculus [196], which provides rules on how programs can be refined, thus allowing to reason about abstract model programs. From this work we get the notion of model program refinement and we include it in the semantics-based relationship predicates as follows: for an action type $A_1$ to be semantics-based conformant to an action type $A_2$, the model program of $A_1$ must be a refinement of the model program of $A_2$.

![Figure 4.5. Selected Function Specification Matches](attachment:image)

Another interesting observation for semantics-base conformance of action types is that depending on the kind of function specification match that we use the relationships may define an equivalence or a partial order for action types. So, focusing only on our selected matches, according to Zaremski Exact pre/post is an equivalence match, Plug-in is a partial order match, while Guarded post and Weak post are neither. For the remaining of the selected matches we show that Exact pred 2 is an equivalence match, while Relaxed plug-in and Guarded gen. pred. are partial order matches with respect to the Exact pred. match.
The discussion so far has focused on action types. We now extend our notions of semantics-based relationships between these types, to notions of semantics-based relationships between service and component behavioural interface types and service and component types. In the case of service behavioural interface types, first we require that their corresponding action types are semantics-based conformant, using one of the function specification matches of Figure 4.5 for their pre-/post-conditions and the program refinement of their model programs. Second, we also need to specify a predicate involving their invariants and the history constraints. For the invariants the work on module specification matching [166], dictates that they should be related using either logical implication or logical equivalence, as follows: for a service behavioural interface type SBI1 to be semantics-based conformant to a service behavioural interface type SBI2, the invariant of SBI1 must logically imply or be equivalent to the invariant of SBI2. Note that in reality, as we mentioned above, the predicate would be a bit more complicated since it would require that an abstraction or simulation function exists that is applied to the invariants before checking the logical operators.

In the case of history constraints Dhara and Leveans [193] say the same approach can be followed. However, this approach runs into problems when additional action types are considered [27, 193]. At this point, we should note that so far all our compatibility/conformance definitions referred to relationships between corresponding elements or features of the related types. This correspondence can be defined as a function that maps the features of one type to features of the other. Although, this function is required to be one-to-one, i.e. no two features of the first type are mapped to the same feature of the second. It does not have to be onto, i.e. there may be additional features of the second types that none of the features of the first type map to. As a result, in the cases where the correspondence function between features is not onto, these unmapped features of the second type are not covered by the compatibility/conformance relationship predicate. In other words, conformant types may have additional features, which may also introduce additional behaviour. For this reason, in behavioural subtyping literature two kinds of behavioural subtyping are identified: strong one and weak ones [193]. The latter follow the approach we already described, i.e. they ignore any additional behaviour introduced by the additional action types. However, the former approaches
require that either no additional behaviour is introduced, i.e. require that the behaviour of the additional actions is explainable in terms of the other ones [197], or that any additional behaviour is also covered by the history constraint, i.e. the logical implications covers all valid computations [193]. Adopting the same distinction, we also refer to strong and weak semantics-based conformance of service behavioural interface types.

The semantics-based conformance of service behaviour interface types can be extended in a straightforward manner to component behavioural interfaces. More specifically, as the component constraints restrict the behaviour of their services (see section 4.2.5), they have to be considered in conjunction with the invariant and history constraints of the relevant services. Then, semantics-based conformance of service behaviour interface types just requires that their corresponding service behavioural interface types including the component constraints are either strong or weak semantics-based conformant. We should note here a couple of points. First, the distinction between provided and required service behavioural interface types is taken into account by considering a plug-in like (see Plug-in function specification match above) relationship between them. More precisely, in order for component behavioural interface type CBI₁ to be semantics-based conformant to component behavioural interface type CBI₂, then CBI₁ provided service behavioural interface types need to be conformant to the corresponding provided service behavioural interface types of CBI₂, while the CBI₂ required service behavioural interface types need to be conformant to the corresponding required service behavioural interface types of CBI₁. In all cases we in fact refer to the service behavioural interface including the component constraints. Note that this plug-in like relationship is in fact a semantic form of the structural contra-/co-variance rule we saw in section 4.5.4. Second, as the behavioural models of required service behavioural interface types are described using model variables that quite often are of MODEL_VARIABLE type, establishing conformance in this case would be in general easier.

In order to extend the above relationships to service and component types, we also need to define semantics-based relationships between their property types. However, our property types do not include descriptions of behaviour. At the same time, for universal and conformant property types their semantics are partly standardised, i.e. they confer the
same meaning in all contexts they are used. Consequently, for these property types we could define notions of semantics-based compatibility/conformance relationships with respect to their names and the domains within in which they are standardised. We provide such relationships in section 4.7. For the rest of the property types use can define such relationships in an ad hoc manner. In both cases, these relationships combined with the relationships defined above for their respective behavioural interface types can be used to define semantics-based compatibility/conformance relationships between service and property types.

4.5.7. Type relationship specification

In the previous section, the type compatibility/conformance relationships have be defined informally. However, for a type manager to handle these relationships they need to be rigorously specified. As we believe that all the above mentioned relationships are relatively straightforward to specify, in this section instead of going through the rigorous specification of each one of them we are just going to present the various elements that these specification must include. These elements are the following:

- **Relationship name.** Relationship names are just strings that follow the same naming scheme as our types. As a result, we can foresee that some of the relationships described above will be universal. These relationships would have to be fundamental for the trading process, as they are required to be known to all traders. Potential candidates relationships may be syntactic-based conformance, with basic name and exact signature conformance, i.e. interface conformance, or a semantic interface conformance, combining the interface conformance with strong semantics-based conformance using exact pre-/post match, etc. Conformant type relationships may also be standardised in particular application domains.

- **The relationship meta-type(s).** This refers to the kind of types that can be related using this particular relation. Note that all the relationships described above are homogenous, i.e. they relate types of the same meta-type. In general, this does not have to be the case. We can define also relationships between different meta-types. For example, we can define relationships that
relate component or service types to their (behavioural) interface types. In fact, we have implicitly used such relationships, when defining the relationships between component or service types.

- **Relationship Properties.** Relationship properties are particularly useful in reducing the amount of checks that need to be carried out every time a new type is introduced. For example, if R is anti-symmetric and T₁ is related with R to T₂ this implies that T₂ cannot be related to T₁ with R. The following properties are of particular interest: Reflexivity, Symmetry, Anti-Symmetry, Transitivity and Surjectivity (onto).

- **Cardinality Constraints.** These constrain the number of types that can be related to each other through a particular type relationship.

- **Runtime/Composite.** This property of the relationships indicates whether they are runtime or development time, and composite and simple ones. As a result, the value is a pair of Boolean values, where the first value indicates if it is runtime or not and the second if it is composite or not.

- **Relationship Semantics.** These are usually expressed by a function that can compute whether particular types are related or not by the relationship. Note that the provided types have to be of the meta-type(s) that the relationship relates. Moreover, a relationship may use another relationships semantics function as part of its calculation. In general, the semantics do not have to be specified mathematically. They could also be described using text. However, in this case the type manager is no longer able to establish in an automated way whether a particular relationship holds.

4.5.8. **Summary**

From a trading point of view the type relationships of interest are those expressing some notion of compatibility/conformance. However, this kind of relationships can be defined in a variety of ways depending on which aspects of the type descriptions we consider. In order to provide a framework within which the various definition can be presented, we started by examining how type relationships are framed in object-oriented typing. Our analysis of the provided framework, though, showed that certain refinements
were necessary in the context of our type model. As a result, we produced the framework depicted in Figure 4.2. Then, based on this framework we described name-based, structure-based, semantics-based and syntactic-based compatibility/conformance relationships. The most interesting of these relationships are the structure-based and semantics-based ones. The former because they build upon the work on function signature matching, while the latter built upon the work on function specification matching and behavioural subtyping. In each case we analysed the related work, and we identified which parts are of particular interest to our type model and how these parts need to be extended to cover the whole spectrum of meta-types that it defines. As a final note we should point out that in the case of component and service interface references their compatibility/conformance is defined in terms of the respective component and service interface types. We concluded our discussion on the type relationships of the model with a description of the various elements that their specifications must comprise of.

4.6. Polymorphism and inheritance

Polymorphism and inheritance are quite common features of type models. In fact, in section 4.2.1 we identified support for some kind of polymorphism and inheritance as one of the requirements for our component type model. In this section we briefly examine how these features are incorporate into the model.

There are three types of polymorphism identified in literature: inclusion polymorphism, parametric polymorphism and ad hoc polymorphism [146]. Starting from the last, it usually refers to polymorphism that is defined in an axiomatic manner through the specification of coercions between types. In order to simplify matters we do not consider at all ad hoc polymorphism. Parametric polymorphism can be supported in two ways, with the introduction of either Any types or some short of template definition of types. In our component type model we include Any types and as a result, we support parametric polymorphism in this way. We should note, though, that using Any types can introduce recursive type definitions, which require some care to be taken when attempting to establish compatibility/conformance relationships between them. Our type model supports inclusion polymorphism primarily through the compatibility/
conformance relationships that define a partial order. These relationships are in fact forms of subtyping. Furthermore, in a similar manner to service trading type models, we also support inclusion polymorphism linked with inheritance.

In general inheritance can take a number of different forms [198], the most common of which are the restriction and extension ones. The former allows child types to restrict the features provided by their parent type, either by making some of them unavailable or by restricting their scope, e.g. a stricter pre-condition. The latter allows child types to extend the features provided by their parent type, either by providing some additional features or by extending their scope, e.g. a less strict pre-condition. Inheritance has been the focus of a lot of heated debates, particular in the context of object-oriented programming languages. However, most of the criticism is due to the fact that in most programming languages inheritance and subtyping are kept separate [199]. In other words, a child type provides very few guarantees of compatibility/conformance. As a result, in our type model we only consider inheritance that is associated with some kind of compatibility/conformance relationship. Note that in this case we are only interested in the relationships that define a partial order on types. Furthermore, we primarily consider inheritance as a mechanism for the incremental description of types. Consequently, when a child type inherits from a parent type its type description only includes the additional features that it adds to those of the parent type.

More specifically, in the case of inheritance in service interface types, the additional features are action interfaces. Note that in this case, the child service interface type is structure-based conformant to the parent. In the case of inheritance in component interface types, in order to also guarantee that the child type is structure-based conformant to the parent type, the additional features can only be provide service interface types. In the case of inheritance in service behavioural interface types, we are referring to a form of specification inheritance [193]. Consequently, we only allow the child type to introduce additional action types, in which case it is guaranteed to be weak semantics-based conformant to the parent type. Similarly, in the case of inheritance in component behavioural interface types, we only allow the child type to introduce additional provided service behavioural interface types. Again, the child type is guaranteed to be weak semantics-based conformant to the parent type. Moreover, in the
case of inheritance in service types, we allow the child type to introduce either additional action types or additional property types. In both cases the child type is guaranteed to be weak semantics-based conformant to the parent type. Finally, in the case of inheritance in component types, we allow the child type to introduce either additional provided service types or additional property types. Again, the child type is guaranteed to be weak semantics-based conformant to the parent type.

In conclusion, in our type model, we support parametric polymorphism through Any types and inclusion polymorphism through subtyping relationships and inheritance. In the case of inheritance we take a fairly restrictive view and consider it a mechanism for the incremental description of types. We define six types of inheritance between, component and service types, component and service behavioural interface types, and component and service interface types.

4.7. Type definition domains

The idea behind the introduction of type definition domains is twofold. First, some of the elements of type descriptions are not particular to a specific type. Instead they are shared by a number of different types. For these elements it seems reasonable to describe them outside any particular type and then allow types to incorporate these descriptions. As we already mention in section 4.2.4, contract types for extra-functional properties and extra-functional property compositions are prime candidates for such treatment. Second, in the area of component software there are a number of efforts underway aiming to standardise application domain specific concepts and types, like business objects [148, 176, 177]. Moreover, in most application domains there are a number of concepts that have a precise meaning and are well understood, i.e. the vocabulary of the application domain. For these concepts Hussmann in [8] claims that it is unreasonable to require their formal specification. These standardised concepts and types are only specific to the particular application domain and not to any particular type. Therefore, they can also be described separately, outside of any particular type. In order to accommodate the above two different aspects, we define two kinds of domains: extra-functional and application specific ones. The former target a particular extra-functional aspect and they define contract types in the sense of QML [25] for this aspect. The latter
target a particular application domain and they primarily define domain specific types and model types, and a domain vocabulary. From this discussion it should clear that the domains in respect to which conformant types are defined are in fact application specific domains.

At this point, we should draw attention to the fact that extra-functional domains can also be used to link type descriptions to attribute-driven contextual composition, as supported by current component platforms (see section 2.3). In this context, the domain instead of defining extra-functional contracts, it defines sets of attributes along with their permissible values that refer to particular extra-functional aspects, i.e. transactional properties, of the respective type. As a result, our notions of types and type compatibility/conformance relationships can also capture the requirements of components in terms of contextual composition.

Moreover, application specific domains can also include information that simplifies behavioural specifications, like behavioural labels [169], and supports reasoning about type relationships, particularly semantics-based ones, like lemmas for proofs. More specifically, behavioural labels are just labels used in the place of particular quite complex predicates, thus simplifying reasoning about semantics-based type relationships. In addition to behavioural labels the domain vocabulary can also be used to simplify behavioural specifications. For example, the concept of interest is well defined within the banking domain. So, the specification of an interesting paying current account service does not need to mathematically specify it in terms of its effect on the account's balance. The domain vocabulary can also be used when naming types specified with respect to the domain. In this case the type name is no longer ambiguous as it is a concept with precise meaning within the particular application domain. Note however that as we cannot check if the meaning intended by the use of a domain concept in a type specification is the same as the one expected by the business domain, we can view such use as placing an obligation to the specifier to guarantee correct semantics. This approach could be strengthened by requiring concepts to be defined as part of an application domain ontology.

The use of domain vocabulary concepts as type names allows us also to define name-based and syntactic-based compatibility/conformance relationships with more
precise semantics. We refer to these relationships as domain enhanced name-based and syntactic-based relationships. In these relationships the predicate relating the names of the types requires not only equality of the respective names, but also that these names are concepts from the same vocabulary domain. In the case of the name-based relationships, since there are no guarantees with respect to the structure of the related types, they are development time relationships. We can extend further the definitions of domain-enhanced relationships to incorporate in their predicates not only the names of their features but also the names of the related types themselves, provided they are also concepts. In this case we are talking about strong domain enhanced relationships. Note that domain-enhanced type compatibility/conformance is a form of ontological conformance as defined in [190].

Each domain has a globally unique name. In order for a type description to use domain information, it needs to import the particular domain. This can do so with the use of the keyword “import” followed by a list of the domain names it wants to import. However, domains can only be used in the definition of component and service related types. Moreover, in the case of service related types although they may import any number extra-functional domains, the can only import a single application specific domain. The reason for this restriction is that as services are cohesive units of functionality they should be targeted towards a single application domain.

Finally, with the introduction of type definition domains type managers have to be associated with one or more domains. For each domain a type manager is associated with it should have available all the elements of the domain specification. Moreover, if the type manager is associated with more than one domain then it could also have a number of “is mapped to” relationships that play the role of semantic bridges between domains. These bridges are particularly useful when handling cross-domain requests.

4.8. Concluding remarks

In this chapter we described the component type model for SECT, which is the main focus of the thesis. The main characteristics of the type model are:

1. Its support for semantic types, i.e. types that include explicit descriptions of behaviour;
2. The introduction of component types as types including sets of provided and required services;
3. Its support for a type naming scheme that takes into consideration standardisation of types, when it distinguishes in addition to private and universal ones, conformant ones too;
4. The provision of rigorous type descriptions, which support the definition of type relationships;
5. The definition of a variety of notions of type compatibility/conformance within an underlying framework that makes their relationships explicit and covers name-based, structure-based, syntactic-based, and semantics-based relationships;
6. The introduction of a number of inheritance relationships between types that support the incremental definition of type descriptions; and
7. The introduction of the notion of type definition domains to support the description of extra-functional properties and application domain concepts and types that can also be utilised in defining ontological type relationships.

However, arguably the most interesting characteristic of the type model is its ability to be extended with the definition of additional type relationships. These relationships can either combine the various dimensions of the type relationship framework in new ways, or take into consideration additional aspects of the type descriptions. For example, although in this thesis we have decided to not consider protocol-level interoperability constraints, these can be seamlessly integrated to the type model to introduce notions of protocol semantics-based compatibility/conformance relationships by adding to the semantics-based relationship predicates for service and component related types, the requirement that their respective protocol constraint are conformant according to some notion protocol conformance (e.g. [200] or [201]).
4.3. Concluding remarks

In this chapter we described the management type model for ESECT, which is the main focus of the thesis. The main characteristics of the type model are:

1. Its support for semantic types, i.e., terms that include detailed descriptions of behavior;}
Chapter 5. SECT component type model evaluation

In the previous chapter we presented the SECT component type model, which is the main focus of this thesis, and in this chapter, we continue to evaluate it. The evaluation is along two dimensions. First, we want to demonstrate that the component type model is quite expressive and able to capture a wide range of variations in the definitions of its various types. Second, referring back to our initial problem statement (see section 1.2), we want to show that by using the component type model SECT not only can bring the notion of service trading into the domain of component-based development, but also has the potential to provide higher quality trading.

In order to achieve the former in section 5.1, we present a number of example type definitions, each demonstrating a different aspect of the component type model. Although, we recognise that these definitions are not fully-fledged “real-world” examples, we believe that they are adequate to support our claim that the SECT component type model is quite expressive. At the same time, so as to achieve the latter in section 5.2, we present an architecture for a SECT-enabled trader. We also show not only that it is possible for the architecture to be realised, but also that certain of its features enable the provision of higher quality trading. However, we should point out that there are still some important technical challenges to be addressed, before the architecture can be fully realised.

5.1. Example SECT component type model type definitions

All the examples in this section are from the banking application domain. The basic idea behind them is that banks could define different services that they offer to their clients as service types, and based on these service types, define a number of component types representing their various products, i.e. certain packages of services. More specifically, we start with the definition of a component type that represents a basic bank account, whose only provided functionality is that of a basic account, i.e. deposit and withdrawal of money. Following the verbose component type description, defined in Definition 4.1, Example 5.1 defines a BankAccount component type. According to Example 5.1, the BankAccount component type provides a single service, defined by
the BasicAccount service type. As the three empty set symbols following the set of provided service types show, it requires no services, and has no component type constraints or component protocol constraints. The fact that it does not define any component constraints and protocol constraints is expected, as their intension is to capture any dependencies between the various services of the component type and in this case there is only a single one. With respect to the property types, as each component is required to have a set of common property types, we have indicatively provided an author common property type. As these property types are universal, we just name them. As the final two empty set symbols indicate, we have not defined any described or named property types.

```
CT_1 = (BankAccount, \{ST\}, \emptyset, \emptyset), where
ST_1 = \{BasicAccount, \{AT_1, AT_2\}, Inv^{BAcc_{BM}}, Con^{BAcc_{BM}}, \emptyset, \emptyset, \emptyset\}
```

```
AT_1 = \{ Deposit, void, \emptyset, \{in, amount, Real\} \}
  \begin{align*}
  &\text{context BasicAccount::deposit} \\
  &\text{pre: amount > 0} \\
  &\text{post: balance = balance@pre + amount} \\
  \end{align*}
```

```
AT_2 = \{ Withdraw, void, \emptyset, \{in, amount, Real\} \}
  \begin{align*}
  &\text{context BasicAccount::withdraw} \\
  &\text{pre: amount > 0 and amount < balance} \\
  &\text{post: balance = balance@pre - amount} \\
  \end{align*}
```

```
Inv^{BAcc_{BM}} = \{(context BasicAccount inv: balance >= 0)\} and
Con^{BAcc_{BM}} = \{(context BasicAccount hist: holder = holder@pre)\}
```

Example 5.1. Verbose definition of component type: BankAccount.

Looking at the definition of the BasicAccount service type and its comprising action types (Deposit and Withdraw) above, we note that their behaviour is specified with respect to the BAcc_{BM} behaviour model. This service behaviour model is
provided in Figure 5.1, in the form of a UML class model. In this class model, holder and balance are the model variables, while deposit and withdraw are the signatures of the comprising action types. We should also note that in the behavioural specifications, we have adopted the conventions that reserved words are written in bold (e.g. `context`), and our proposed extensions to OCL are written in italicised bold (e.g. `hist`). Furthermore, in the case of history constraints the `@pre` is also italicised as we have extended its meaning, see section 4.4.1. The meaning of the history constraint is that the holder of an account is fixed, i.e. cannot change over time. Finally, the empty curly brackets at the end of the action types represent the fact that they have no associated model programs.

![Figure 5.1. BasicAccount service behaviour model (BAcc_BM).](image)

\[
AT_i = \begin{pmatrix}
\text{Deposit}, \text{void}, \emptyset, \{(\text{in}, \text{amount}, \text{Real})}\}, \\
\{\text{context} \text{BasicAccount}: \text{deposit} \}
\end{pmatrix}
\]

**Example 5.2. Action type definition: Deposit.**

We can now break down the definition of the BankAccount component type into its constituent types, in order to provide example definitions of the various kinds of
types of the component type model. So, starting from action-related types, Example 5.2 shows the definition of action type Deposit according to Definition 4.5, while Example 5.3 shows the corresponding action interface type definition, i.e. the action signature, according to Definition 4.12, namely Deposit_I. Note that the empty set symbol in both definitions denotes the fact that Deposit and Deposit_I do not have any exceptional terminations, i.e. exceptions.

\[ \text{AIT}_1 = \{ \text{Deposit}_I \}, \text{void}, \emptyset, \{ (\text{in}, \text{amount}, \text{Real}) \} \]

Example 5.3. Action interface type definition: Deposit_I.

\[ \text{ST}_1 = \{ \text{BasicAccount}, \{ \text{AT}_1, \text{AT}_2 \}, \text{Inv}^{\text{BasicAccount_BI}}, \text{Con}^{\text{BasicAccount_BI}}, \emptyset, \emptyset, \emptyset \} \]

\[ \begin{align*}
\text{AT}_1 &= \{ \text{Deposit}, \text{void}, \emptyset, \{ (\text{in}, \text{amount}, \text{Real}) \},
\begin{align*}
\text{context} & \text{BasicAccount}::\text{deposit} \\
\text{pre} & : \text{amount} > 0 \\
\text{post} & : \text{balance} = \text{balance}@\text{pre} + \text{amount}
\end{align*} \\
\text{AT}_2 &= \{ \text{Withdraw}, \text{void}, \emptyset, \{ (\text{in}, \text{amount}, \text{Real}) \},
\begin{align*}
\text{context} & \text{BasicAccount}::\text{withdraw} \\
\text{pre} & : \text{amount} > 0 \text{ and } \text{amount} < \text{balance} \\
\text{post} & : \text{balance} = \text{balance}@\text{pre} - \text{amount}
\end{align*} \\
\text{Inv}^{\text{BasicAccount_BI}} &= \{ \{ \text{context} \text{BasicAccount inv: balance} \geq 0 \}, \text{and} \\
\text{Con}^{\text{BasicAccount_BI}} &= \{ \{ \text{context} \text{BasicAccount hist: holder} = \text{holder}@\text{pre} \} \}
\end{align*} \]

Example 5.4. Verbose service type definition: BasicAccount.

Similarly, Example 5.4 shows the verbose definition of the BasicAccount service type according to Definition 4.3. Note that three empty set symbols denote the fact that BasicAccount does not have any protocol constraints, and named and described property types, respectively. Example 5.5 shows the corresponding service behavioural interface type definition according to Definition 4.9, namely BasicAccount_BI, and Example 5.6 the corresponding service interface type
definition according to Definition 4.11, namely BasicAccount_I. Note also that while the service behavioural interface type and service type incorporate the corresponding action type definitions, the service interface type incorporates the action interface types instead.

\[
SBI = (\text{BasicAccount}_\text{BI}, \{AT_1, AT_2\}, Inv^{\text{Bank BMI}}, Con^{\text{Bank BMI}}, \emptyset)
\]

where \(AT_i =\)

\[
\begin{align*}
\text{Deposit, void, } &\emptyset, \{(\text{in, amount, Real})\}, \\
\text{context BasicAccount::deposit} &, \text{pre: amount > 0} \\
\text{context BasicAccount::deposit} &, \text{post: } balance = balance@pre + amount
\end{align*}
\]

\[
\text{Withdraw, void, } &\emptyset, \{(\text{in, amount, Real})\}, \\
\text{context BasicAccount::withdraw} &, \text{pre: amount > 0 and amount < balance} \\
\text{context BasicAccount::withdraw} &, \text{post: } balance = balance@pre - amount
\]

\[
Inv^{\text{Bank BMI}} = \{(\text{context BasicAccount inv: balance }\geq 0)\}, \text{and} \\
Con^{\text{Bank BMI}} = \{(\text{context BasicAccount hist: holder = holder@pre})\}
\]

Example 5.5. Service behavioural interface type definition: BasicAccount_BI.

\[
SIT = (\text{BasicAccount}_\text{I}, \{AIT_1, AIT_2\}) \text{ where} \\
AIT_1 = (\text{Deposit}_\text{I, void, } \emptyset, \{(\text{in, amount, Real})\}) \\
AIT_2 = (\text{Withdraw}_\text{I, void, } \emptyset, \{(\text{in, amount, Real})\})
\]

Example 5.6. Service interface type definition: BasicAccount_I.

Likewise in the case of the component type BankAccount, Example 5.7 shows the corresponding component behavioural interface type definition according to Definition 4.8, namely BankAccount_BI. The three empty set symbols denote the fact that BankAccount_BI has no required service types, component constraints, and component protocol constraints respectively. Example 5.8 show the corresponding component interface type definition according to Definition 4.10, namely
BankAccount_I. Note that in this case, the definition incorporates the corresponding
service interface type BasicAccount_I.

\[
\text{CBIT}_1 = (\text{BankAccount}_\text{BI}, S\text{BIT}_1, \emptyset, \emptyset, \emptyset), \quad \text{where}
\]

\[
S\text{BIT}_1 = \{\text{BasicAccount}_\text{BI}, \{A\bar{T}_1, A\bar{T}_2\}, \text{Inv}^{\text{BAcc}_{\text{BM}}}, \text{Con}^{\text{BAcc}_{\text{BM}}}, \emptyset\}\text{and}
\]

\[
A\bar{T}_1 = \left\{
\begin{array}{l}
\text{Deposit}, \emptyset, \emptyset, \{(\text{in}, \text{amount}, \text{Real})\}, \\
\quad \text{context BasicAccount::deposit} \\
\quad \text{pre: amount > 0} \\
\quad \text{post: balance = balance@pre + amount}
\end{array}
\right\}
\]

\[
A\bar{T}_2 = \left\{
\begin{array}{l}
\text{Withdraw}, \emptyset, \emptyset, \{(\text{in}, \text{amount}, \text{Real})\}, \\
\quad \text{context BasicAccount::withdraw} \\
\quad \text{pre: amount > 0 and amount < balance} \\
\quad \text{post: balance = balance@pre - amount}
\end{array}
\right\}
\]

\[
\text{Inv}^{\text{BAcc}_{\text{BM}}} = \{(\text{context BasicAccount inv: balance >= 0})\} \text{and}
\]

\[
\text{Con}^{\text{BAcc}_{\text{BM}}} = \{(\text{context BasicAccount hist: holder = holder@pre})\}
\]

**Example 5.7. Component behavioural interface type definition: BankAccount_BI.**

\[
\text{CIT}_1 = (\text{BankAccount}_\text{I}, \{S\text{IT}_1\}, \emptyset) \text{where}
\]

\[
S\text{IT}_1 = (\text{BasicAccount}_\text{I}, \{A\bar{T}_1, A\bar{T}_2\}), \text{and}
\]

\[
A\bar{T}_1 = (\text{Deposit}_\text{I}, \emptyset, \emptyset, \{(\text{in}, \text{amount}, \text{Real})\}) \text{and}
\]

\[
A\bar{T}_2 = (\text{Withdraw}_\text{I}, \emptyset, \emptyset, \{(\text{in}, \text{amount}, \text{Real})\})
\]

**Example 5.8. Component interface type definitions: BankAccount_I.**

As we mentioned in section 4.4, for component and service types in addition to
their verbose definitions (Definition 4.1 and Definition 4.3), the component type model
also supports their compact definition, see Definition 4.2 and Definition 4.4, respectively.
These definitions are expressed in terms of the corresponding component and service
behavioural interface types. Consequently, Example 5.9 shows the compact definition of
the BankAccount component type in terms of the BankAccount_BI component
behavioural interface type (see Example 5.7), while Example 5.10 shows the compact
definition of the BasicAccount service type in terms of the BasicAccount_BI service
behavioural interface type (see Example 5.5). A comparison of Example 5.9 and Example
5.10 to Example 5.1 and Example 5.4 fully justifies the characterisation of these
definitions as compact and verbose, respectively.

\[ CT_1 = \{ \text{BankAccount, BankAccount\_BI, } \langle \text{author, String}, ... \rangle \} \]

Example 5.9. Compact component type definition: BankAccount.

\[ ST_1 = \{ \text{BasicAccount, BasicAccount\_BI, } \emptyset, \emptyset \} \]

Example 5.10. Compact service type definition: BasicAccount.

5.1.1. Looking into the relationships between example types

Having presented examples of all the various kinds of types of the SECT
component type model, let us now present some additional examples that demonstrate
variations between types and type relationships. For this purpose in Example 5.11 we
present the definition of a component type providing a single service of an interest paying
account, namely IPBankAccount. This component type definition is with respect to
the service behavioural model in Figure 5.2. A closer examination of the elements of the
component type definition reveals that the definitions of action types IPBasicAccount\_Deposit and IPBasicAccount\_Withdraw look identical to
those of BasicAccount\_Deposit and BasicAccount\_Withdraw respectively. Nevertheless, they are not as their behaviour is specified with respect to different service
behavioural models (see Figure 5.1 and Figure 5.2, respectively). At the same time, it is
not difficult to show that the action types are respectively equivalent, in terms of name-
based, structure-based (using exact signature matching), syntactic-based and even
semantics-based (using Exact pre/post specification matching) equivalence. Moreover, the history constraints of the service types BasicAccount and
IPBasicAccount are also equivalent, while the invariant of IPBasicAccount implies that of BasicAccount. Taking all these into consideration we can infer that
IPBasicAccount is in fact a behavioural subtype of BasicAccount in the sense
advocated in [217, 197]. Furthermore, it is a strong behavioural subtype [193], as their
history constraints are equivalent despite the addition of the extra methods. The same relationship also holds for the component types IPBankAccount and BankAccount, as the two service types are the only ones they provided and they only incorporate common property types that by definition are the same for all component types.

![Figure 5.2. IPBasicAccount service behavioural model (IBAcc_BM).](image)

In Example 5.12 we present the definition of a component type providing a single service of a fixed rate interest paying account, namely FRBankAccount. Its definition is with respect to the behavioural model presented in Figure 5.3. Following a similar line reasoning as above, it is easy to show that FRBankAccount is also a strong behavioural subtype of BankAccount. However, it is also interesting to point out that IPBankAccount is a weak behavioural subtype of FRBankAccount. In order to understand why this is the case we need to first remind the reader that the difference between a strong and a weak behavioural subtype is whether the history constraint of the subtype implies that of the supertype for all valid computations, including those carried out by the additional actions it introduces [193]. In the case of service type IPBasicAccount, if we do not consider its additional action type ChangeIntRate then it satisfies the history constraint of the FRBasicAccount service type and can thus be considered its behavioural subtype. Note that with respect to all their other elements the two service types are in fact equivalent. As a result, component type IPBankAccount can be considered a weak behavioural subtype of FRBankAccount.
Example 5.11. IPAccount component type definition.
CT\textsubscript{3} = (FRBankAccount, \{ST\textsubscript{3}\}, \emptyset, \emptyset, \{(author, String)\}, \emptyset, \emptyset, \{\}
where

\[\begin{align*}
ST\textsubscript{3} &= \{FRBasicAccount, \{AT\textsubscript{7}, AT\textsubscript{8}, AT\textsubscript{9}\},
\text{Inv}_{FM}^{FRBasicBM}, Con_{FM}^{FRBasicBM}, \emptyset, \emptyset, \emptyset\}, \text{and}
\end{align*}\]

\[\begin{align*}
AT\textsubscript{7} &= \{\text{Deposit, void, } \emptyset, \{(in, amount, Real)\},
\text{context FRBasicAccount::deposit },
\text{pre: amount} > 0
\text{context FRBasicAccount::deposit },
\text{post: balance} = \text{balance}_{@pre} + \text{amount}\}, \{\}
\end{align*}\]

\[\begin{align*}
AT\textsubscript{8} &= \{\text{Withdraw, void, } \emptyset, \{(in, amount, Real)\},
\text{context FRBasicAccount::withdraw },
\text{pre: amount} > 0 \text{ and amount} < \text{balance}
\text{context FRBasicAccount::withdraw },
\text{post: balance} = \text{balance}_{@pre} - \text{amount}\}, \{\}
\end{align*}\]

\[\begin{align*}
AT\textsubscript{9} &= \{\text{PayInterest, void, } \emptyset, \emptyset,
\text{context FRBasicAccount::pay_interest },
\text{pre: balance} > 0
\text{context FRBasicAccount::pay_interest },
\text{post: balance} = \text{balance}_{@pre} \times (1 + \text{int_rate})\}, \{\}
\end{align*}\]

\[\begin{align*}
\text{Inv}_{FM}^{FRBasicBM} &= \{\text{context FRBasicAccount }
\text{inv: balance} \geq 0 \text{ and } 0 < \text{int_rate} \text{ and } \text{int_rate} \leq 1\}\}
\end{align*}\]

and

\[\begin{align*}
\text{Con}_{FM}^{FRBasicBM} &= \{\text{context FRBasicAccount }\}
\text{hist: holder} = \text{holder}_{@pre} \text{ and }
\text{int_rate} = \text{int_rate}_{@pre}\}
\end{align*}\]

Example 5.12. FRBankAccount component type definition.

In Example 5.13 we present the definition of a component type providing a single service of a guaranteed rate interest paying account, namely GRBankAccount. It is interesting to note that this component type definition is with respect to the previously defined behavioural model in Figure 5.2. As a result, the definitions of the action types
Deposit, Withdraw and PayInterest of service type GRBasicAccount are really identical to those with the same names of service type IPBasicAccount. This is the reason why we refer to them using the same variables (AT₃, AT₄, and AT₅). Following similar reasoning as above, we can easily show that GRBankAccount is a strong behavioural subtype of BankAccount, and that GRBankAccount is a weak behavioural subtype of FRBankAccount.

![FRBasicAccount behavioural model (FBAcc_BM).](image)

However, the main reason for including Example 5.13 is to examine the relationship between GRBankAccount and IPBankAccount. Since, as we mentioned above the definition of the first three of their action types are identical, we focus our attention on the relationship between action types IncreaseIntRate and ChangeIntRate. As both action types have the same signature, they are structure-based equivalent using the exact signature match. Regarding their semantics, ChangeIntRate is semantics-based conformant to IncreaseIntRate using Plug-in specification match. This relationship though cannot be further utilised to show that IPBasicAccount is a behavioural subtype of GRBasicAccount, as the history constraint of IPBasicAccount does not imply that of GRBasicAccount. At the same time, IncreaseIntRate is semantics-based conformant to ChangeIntRate using Weak post specification match. This means that provided that we can ensure that the IncreaseIntRate pre-condition holds, we can use it in
the place of ChangeIntRate. It is easy to show that this kind of relationship also holds between GRBasicAccount and IPBasicAccount, or even between GRBankAccount and IPBankAccount. We should note that as we mentioned in section 4.5.6, this kind of type relationship is a composite one and thus not immediately usable.

Figure 5.4. OBasicAccount behavioural model (OAcc_BM).

In Example 5.14 we present the definition of a component type providing a single service of an account with an overdraft, namely OBankAccount. Its definition is with respect to the behavioural model presented in Figure 5.4. Following a similar line reasoning as above for their Withdraw action types, we can see that they both have the same signature. As a result they are structure-based equivalent using the exact signature match. Regarding their semantics, OBasicAccount.Withdraw is semantics-based conformant to BasicAccount.Withdraw using Plug-in specification match. In the same way as above, this relationship cannot be further utilised, as in this case the invariant of OBasicAccount does not imply that of BasicAccount. At the same time, BasicAccount.Withdraw is semantics-based conformant to OBasicAccount.Withdraw using Weak post specification match, which means that this kind of relationship also holds between BasicAccount and OBasicAccount, or even between BankAccount and OBankAccount.
In Example 5.15 we present the definition of a component type providing a single service of a credit account, namely CreditAccount. Its definition is with respect to the behavioural model presented in Figure 5.5. The interesting point about this example is that CreditAccount and OBankAccount are both structure-based equivalent using exact signature match, and semantics-based equivalent using Exact pre/post specification match. In order to demonstrate this, we need to establish a correspondence between their respective action types. So, the Charge action type maps to Withdraw and the Credit maps to Deposit. In this case, it is easy to see that their corresponding signatures are in fact identical. However, in order to establish the equivalence of their behavioural specifications we first need to define a mapping between their respective behavioural models (see Figure 5.4 and Figure 5.5). Therefore, we map limit to overdraft and balance to -balance, i.e. we note that the balance plays the same role in both models with the difference that in the BasicCredit model its role is negative. Making the appropriate substitutions in the behavioural descriptions they become in fact identical, establishing the semantics-based equivalence using Exact pre/post specification match of the corresponding action types. The same is true for the invariants of the corresponding service types, which are also semantics-based equivalent using Exact pre/post specification match, consequently the same is the case for the component types too.
\[ CT_4 = (GRBankAccount, \{ST_4\}, \emptyset, \emptyset, \{(author, String)\}, \emptyset, \emptyset), \]

where \( ST_4 = \{GRBasicAccount, \{AT_1, AT_4, AT_5, AT_{10}\}\}, \text{ and} \]

\[
\begin{align*}
\text{Deposit, } &\text{void, } \emptyset, \{(\text{in, amount, Real})\}, \\
AT_1 = &\text{context IPBasicAccount::deposit }\text{ pre: } \text{amount}>0, \\
&\text{post: balance = balance@pre+amount}\} \\
\text{Withdraw, } &\text{void, } \emptyset, \{(\text{in, amount, Real})\}, \\
AT_4 = &\text{context IPBasicAccount::withdraw }\text{ pre: } \text{amount}>0 \text{ and amount}<\text{balance}, \\
&\text{post: balance = balance@pre-amount}\} \\
\text{PayInterest, } &\text{void, } \emptyset, \emptyset, \\
AT_5 = &\text{context IPBasicAccount::pay_interest }\text{ pre: } \text{balance}>0, \\
&\text{post: balance = balance@pre*(1+int_rate)}\} \\
\text{IncreaseIntRate, } &\text{void, } \emptyset, \{(\text{in, new_rate, Real})\}, \\
AT_{10} = &\text{context IPBasicAccount::change_int_rate }\text{ pre: } \text{0<new_rate and new_rate<=1 and } \\
&\text{new_rate}>\text{int_rate}, \\
&\text{post: int_rate = new_rate}\} \\
Inv^{IBAccBM} = &\text{context IPBasicAccount inv: balance>=0 and } \\
&\text{0<int_rate and int_rate<=1} \\
\text{and Con}^{IBAccBM} = &\text{context IPBasicAccount }\text{ hist: holder=holder@pre and } \\
&\text{int_rate>int_rate@pre}\}
\]

Example 5.13. GRBankAccount component type definition.
\[
CT_i = (\text{OBankAccount}, \{ST_i\}, \emptyset, \emptyset, \{(\text{author, String})\}, \emptyset, \emptyset), \text{where}
\]

\[
ST_i = (\text{OBasicAccount}, \{AT_{i1}, AT_{i2}\}, Inv^{\text{OAcc-BM}}, Con^{\text{OAcc-BM}}, \emptyset, \emptyset, \emptyset)
\]

and

\[
\begin{align*}
AT_{i1} &= \left(\text{context OverdraftAccount} : \text{deposit pre: amount} > 0\right), \\
&\quad \left(\text{context OverdraftAccount} : \text{deposit post: balance} = \text{balance} @ \text{pre} + \text{amount}\right), \\
AT_{i2} &= \left(\text{context OverdraftAccount} : \text{withdraw pre: amount} > 0 \text{ and amount} \leq \text{balance} + \text{overdraft}\right), \\
&\quad \left(\text{context OverdraftAccount} : \text{withdraw post: balance} = \text{balance} @ \text{pre} - \text{amount}\right)
\end{align*}
\]

\[
Inv^{\text{OAcc-BM}} = \left(\text{context OverdraftAccount} \right)
\]

\[
\begin{align*}
\text{inv: balance} + \text{overdraft} \geq 0 \text{ and overdraft} \geq 0\right)\}
\text{and}
Con^{\text{OAcc-BM}} = \left(\text{context OverdraftAccount} \right)
\]

\[
\text{hist: holder} = \text{holder} @ \text{pre}\}
\]

Example 5.14. OBanl Account component type definition.

\[
CT_e = (\text{CreditAccount}, \{ST_e\}, \emptyset, \emptyset, \{(\text{author, String})\}, \emptyset, \emptyset), \text{where}
\]

\[
ST_e = (\text{BasicCredit}, \{AT_{e1}, AT_{e2}\}, Inv^{\text{BCredit-BM}}, Con^{\text{BCredit-BM}}, \emptyset, \emptyset, \emptyset)
\]

and

\[
\begin{align*}
AT_{e1} &= \left(\text{context BCredit} : \text{credit pre: amount} > 0\right), \\
&\quad \left(\text{context BCredit} : \text{credit post: balance} = \text{balance} @ \text{pre} - \text{amount}\right), \\
AT_{e2} &= \left(\text{context BCredit} : \text{charge pre: amount} > 0 \text{ and amount} \leq \text{limit} - \text{balance}\right), \\
&\quad \left(\text{context BCredit} : \text{charge post: balance} = \text{balance} @ \text{pre} + \text{amount}\right)
\end{align*}
\]

\[
Inv^{\text{BCredit-BM}} = \left(\text{context BCredit} \right)
\]

\[
\text{inv: balance} \leq \text{limit} \text{ and limit} \geq 0\}
\text{and}
Con^{\text{BCredit-BM}} = \left(\text{context BCredit} \right)
\]

\[
\text{hist: holder} = \text{holder} @ \text{pre}\}
\]

Example 5.15. CreditAccount component type definition.
5.1.2. Looking into more elaborate behavioural specifications

So far, we have not presented any example type definition where the incorporated action types require model programs for their behavioural specification. In Example 5.16 we present such a service type, namely CardAwareAccount. The idea is that this service type enhances the BasicAccount one (see Example 5.10) with the addition of notifications to card services for any change in the accounts balance. Card services support debit and credit cards. The definition of CardAwareAccount is with respect to the behavioural model presented in Figure 5.6. It is interesting to note that this is the first example where the behavioural model is not only using basic model variable types. Instead, it also includes a constructed model variable type Card. This is why the provided diagram includes two classes associated with each other. Regarding the provided model programs, their most notable feature is the use of the iterate operation on collections. As its italicised name indicates, this operation is not used with the usual OCL meaning. Instead it has been modified to accommodate an iteration construct in model programs. This is why in contrast to the usual OCL operation it does not include an accumulator part. The meaning of the modified operator is that we go over all the elements of the collection executing the expression, in this case invoking their balance_change action. Finally, we should also point out that the CardAwareAccount plays the role of the view in a typical model-view-controller pattern [114], and its definition was inspired by [5].

![CardAwareAccount behavioural model (CAcc_BM).](image)

Figure 5.6.
\[ ST_7 = \left( \text{CardAwareAccount}, \{ AT_{i5}, AT_{i6}, AT_{i7}, AT_{i8} \} \right), \text{ where} \]

\[
\begin{align*}
AT_{i5} &= \left( \begin{array}{l}
\text{AddCard, void, } \emptyset, \{ \text{in, card, Card} \}, \\
\text{(context CAAccount::add_card pre: card } \not= \text{ null)}, \\
\text{(context CAAccount::add_card),} \\
\text{(post: cards-} \rightarrow \text{ includes(card))},
\end{array} \right) \\
\text{RemoveCard, void, } \emptyset, \{ \text{in, card, Card} \}, \\
\text{(context CAAccount::remove_card pre: card } \not= \text{ null and cards-} \rightarrow \text{ includes(card))}, \\
\text{(context CAAccount::remove_card),} \\
\text{(post: cards-} \rightarrow \text{ excludes(card))},
\end{align*}
\]

\[
\begin{align*}
AT_{i6} &= \left( \begin{array}{l}
\text{Deposit, void, } \emptyset, \{ \text{in, amount, Real} \}, \\
\text{(context CAAccount::deposit pre: amount } > 0), \\
\text{(context CAAccount::deposit),} \\
\text{(post: balance = balance@pre + amount),} \\
\text{updating = true;} \\
\text{(cards-} \rightarrow \text{ iterate(c:Card, c.balance_change(balance))),} \\
\text{updating = false;}
\end{array} \right) \\
\text{Withdraw, void, } \emptyset, \{ \text{in, amount, Real} \}, \\
\text{(context CAAccount::withdraw pre: amount } > 0 \text{ and amount } < \text{ balance),} \\
\text{(context CAAccount::withdraw),} \\
\text{(post: balance = balance@pre - amount),} \\
\text{updating = true;} \\
\text{(cards-} \rightarrow \text{ iterate(c:Card, c.balance_change(balance))),} \\
\text{updating = false;}
\end{align*}
\]

\[
\begin{align*}
\text{Inv}_{\text{CAAcc_BM}} &= \{ \text{(context CAAccount inv: balance } \geq 0) \}, \text{ and} \\
\text{Con}_{\text{CAAcc_BM}} &= \{ \text{(context CAAccount hist: holder = holder@pre)} \}
\end{align*}
\]

Example 5.16. CardAwareAccount service type specification.
Before we continue to present our final example, we should point out that as the CardAwareAccount service type extends the BasicAccount one, it looks like a prime candidate for using service type inheritance for its definition. Keeping in mind that as we discussed in section 4.6 we use inheritance for the incremental definition of types, then CardAwareAccount would only have to define the features it adds to the BasicAccount. More specifically, it would have to define the AddCard and RemoveCard action types as well as the model programs it associated to the already defined Withdraw and Deposit action types. Following a similar approach as [193], the complete type definition would be the combination of the parent type definition and the additional features, in which case we have the problem of how to deal with the model programs. We take the approach that when there are model programs we consider them in conjunction with the postcondition of the action type. As a consequence, our inheritance is in fact the same kind of specification inheritance described in [193], and guarantees that CardAwareAccount is a behavioural subtype of BasicAccount using the definition of Dhara and Leavens [5], i.e. employing the Guarded gen. pred. specification match.

5.1.3. A final more complete example type definition

Our final example provides the definition of a component type that provides a debit card service and requires an account service that is able to notify the card of any changes in its balance. Example 5.17 and Example 5.18 provide the definition of the Switheard component type according to the component behaviour model provided in Figure 5.7. Note that the split of the definition into parts is only to make easier their typesetting. Regarding the behavioural model in Figure 5.7, we should note that a separate class in the diagram represents each service type of the component type, and that all classes are included within a package representing the component itself. Moreover, the behaviour model of the required service type CAAccount uses the MODEL_VARIABLE type for its card variable. This is to indicate that it is not reasonable for the SwitchCard behaviour model to know the exact form (type) of the card model variable (see section 4.4.1). Regarding the type definition itself, the first thing to note is the existence of a component constraint, which requires that the daily
debit car limit does not exceed the balance of the account, i.e. with a debit card you cannot use more money than you have available. The next thing to note is that the definition of the provided service type is a lot more detailed than that of the required one. The reason for this is that we do not want to unnecessarily restrict the number of suitable service types by describing aspects of their definitions that are not essential for the component type. This becomes particularly clear, if we compare the definition of the CardAccount to the definition of the CardAwareAccount provided in Example 5.16, which aims to provide the same kind of functionality. From this comparison it should also be clear that the CardAccount and DebitCard service types are parts of a model-view-controller pattern design, in which the CardAccount plays the role of the model and the DebitCard the combined role of a view and a controller. Finally, it is interesting to point out that the pre-/post-conditions and model programs of the action types of the provided service type specify in quite a lot of detail the behaviour of the required service type.

Figure 5.7. SwitchCard component behaviour model (SCard_BM).

\[
CT_7 = \left( \text{SwitchCard}, \{ ST_8 \}, \{ ST_9 \}, \right), \text{where} \\
\left( \text{DebitCard}, \{ AT_{19}, AT_{20}, AT_{21}, AT_{22} \} \right), \text{and} \\
ST_8 = \left( \text{Inv}_{\text{SCard_BM}} \text{St}_{\text{7}}, \text{Con}_{\text{SCard_BM}} \text{St}_{\text{7}}, \emptyset, \emptyset, \emptyset \right), \text{and} \\
ST_9 = \left( \text{CardAccount}, \{ AT_{23}, AT_{24}, AT_{25}, AT_{26} \}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \right)
\]

Example 5.17. SwitchCard component type definition.
\[ \text{Inv}^{\text{SCard BM}}_{S_{t_4}} = \begin{cases} \text{context SwitchCard::DCard} \\
\text{inv: } \text{day\_limit} \geq 0 \text{ and day\_limit} \leq \text{max\_limit} \\
\quad \text{and day\_balance} \geq 0 \text{ and day\_balance} \leq \text{day\_limit} \end{cases} \]

and \[ \text{Const}^{\text{SCard BM}}_{S_{t_4}} = \begin{cases} \text{context SwitchCard::DCard} \\
\text{hist: holder=holder@pre} \end{cases} \]

\[ A_{T_{19}} = \begin{cases} \text{ActivateCard, void, } \emptyset, \emptyset, \\
\text{(context SwitchCard::DCard::activate\_card)} \\
\text{pre: account} \neq \text{null} \\
\text{(context SwitchCard::DCard::activate\_card)} \\
\text{post: account.cards->includes(self)} \\
\{\text{account.add\_card(self)}\} \end{cases} \]

\[ A_{T_{20}} = \begin{cases} \text{CancelCard, void, } \emptyset, \emptyset, \\
\text{(context SwitchCard::DCard::cancel\_card)} \\
\text{pre: account} \neq \text{null and} \\
\text{(account.cards->includes(self))} \\
\text{(context SwitchCard::DCard::cancel\_card)} \\
\text{post: account.cards->excludes(self)} \\
\{\text{account.remove\_card(self)}\} \end{cases} \]

\[ A_{T_{21}} = \begin{cases} \text{Charge, void, } \emptyset, \{\text{in, amount, Real}\}, \\
\text{(context SwitchCard::DCard::charge)} \\
\text{pre: amount} > 0 \text{ and account} \neq \text{null and} \\
\text{(amount} \leq \text{day\_limit} - \text{day\_balance}) \\
\text{(context SwitchCard::DCard::charge)} \\
\text{post: day\_balance=day\_balance@pre+amount} \\
\{\text{account.withdraw(amount)}\} \end{cases} \]

\[ A_{T_{22}} = \begin{cases} \text{ChangeDayLimit, void, } \emptyset, \{\text{in, balance, Real}\}, \\
\text{(context SwitchCard::DCard::change\_dLimit)} \\
\text{pre: balance} \geq 0 \text{ and account} \neq \text{null and} \\
\text{(account.updating)} \\
\text{(context SwitchCard::DCard::change\_dLimit)} \\
\text{post: if balance}\geq\text{max\_limit} \\
\text{then day\_limit} = \text{max\_limit} \\
\text{else day\_limit} = \text{balance }\text{endif} \end{cases} \]
Example 5.18. SwitchCard component type definition (cont.).

5.1.4. Concluding remarks

In this section we have presented a number of example type definitions that demonstrate the ability of the SECT component type model to capture a wide range of variations in component behaviour. Our discussion on the syntactic aspects of the provided type definitions has been quite limited as these aspects are well understood and have also been explored by others (see [4] for example). As a result, our emphasis has
been primarily on the semantics of the provided types, in our opinion the most interesting and challenging aspect of the component type model. Particular attention has been placed on the use of model programs in behavioural specifications, arguably the most innovative aspect of the SECT component type model. In the light of the example model programs we also demonstrated how the UML/OCL extensions discussed in section 4.4.1, could be introduced. At this point we should note that, as the following section will demonstrate, it is semantic features like these ones, which the component type model introduces, that play a central role in enabling it to support a higher quality trading. In general, we believe that despite their limited number the presented examples adequately demonstrate the expressiveness of the proposed SECT component type model.

5.2. An architecture for a SECT-enabled trader

In the previous section we presented a number of examples that demonstrate the ability of the component type model to support the detailed expression of type characteristics in a manner that can capture precisely a wide range of variations between them. This presentation raises the question of how could a trader utilise the component type model in order to provide a higher quality trading than current service traders. As we mentioned in section 1.2, in determining the quality of the trading process, we regard trading as a retrieval process and as such we consider the information retrieval measures of precision and recall. So, in this section we present an architecture for a trader that exploits the particular characteristics of the component type model for SECT, i.e. a SECT-enabled trader, and we demonstrate how the architecture has the potential to improve service trading in terms of both precision and recall.

In section 3.2 we saw that in service trading, the trader is primarily responsible for two functions, namely type management and offer management, while the trading process combines the two, in order to match requests for particular service types with offers of these or conformant types. At this level, the operation of the SECT-enabled trader is similar to a service trader. As a result, the architecture for the SECT-enabled trader is based on a type manager, an offer manager, and a request processor, which is responsible for carrying out the trading process itself. What makes the trader SECT-enabled is the fact that its type manager supports the SECT component type model.
presented in Chapter 4. This has a number of implications for all the components of the architecture, which we will examine in turn in the following sections. However, it also has an important implication for the architecture itself, which in order to carry out the trading process requires besides the above components also the support of a composition facility. The role of the composition facility is primarily to support composite offer matches that result from composite type compatibility/conformance relationships.

Figure 5.8 provides a high-level view of the SECT-enabled trader architecture. According to this figure, the trading process starts by sending a request for component offers of a particular type to the request processor. The request processor, first contacts the type manager in order to find other types, conformant to the requested one, as offers of these types may also be returned to the requester. As in this phase of the request processing we in fact expand the number of types, offers of which are going to be considered. As a result, we refer to this phase as the query expansion phase. Having received the number of types from the type manager, then the request processor contacts the offer manager to get any available offers for these types. The provided offers are either simple, i.e. involving a single component that is immediately usable, or composite, i.e. either involving more than one components or requiring some adaptation before use. The simple offers can be immediately returned to the requester, while the composite ones need to be forwarded to the composition facility along with instructions on what kind of adaptation and/or composition they require. The composition facility will carry out the required adaptation and/or composition and will provide the request processor with the
resulting offers. These offers can now be returned to the requester. It is interesting to note that from the requester's point of view composite and simple offers are indistinguishable.

We should point out that in Figure 5.8, we do not have a clear trader boundary showing which of the components (request processor, type manager, offer manager, and composition facility) are internal and which are external to the trader. The reason for this is that we can consider a whole range of alternative implementations of the architecture. With the exception of the request processor that has to be an internal component of any implementation, as it encapsulates the flow of control of the trading process, all other components can either be internal or external. We should also note that although we refer to the elements of the architecture as components this does not mean that they follow our adopted definition of software components, Definition 2.1. As a result, they should not be considered as units of composition. In fact, as we are going to see in the following section for the type manager it probably makes sense to consider part of its functionality as internal and part as external to the trader.

In the following sections we elaborate the above high-level architecture by looking in detail into each one of its main components. However, before we continue to examine these components, we need to point out that although the trader's type manager maintains all the different kinds of types and type relationships described in Chapter 4, the trading process is restricted exclusively to component offers and component types, and utilises only relationships between component types that capture their semantics, i.e. semantics-based and domain enhanced name-based ones. Finally, we conclude our presentation of the trader architecture with a discussion on how it can support the development of the kind of enterprise information systems that provided the motivation for this thesis (see section 1.1).

5.2.1. Request processor

As we mentioned above, the request processor is the component of the architecture that carries out the trading process. As a result, its role within the trader architecture is twofold. On one hand, it provides a query interface that clients can use to submit their requests for component offers. On the other hand, it coordinates the interaction between the other components of the trader architecture. As the trading
process relies on certain kinds of type compatibility/conformance relationships supported by the SECT component type model (see above), the way in which the request processor carries out its role depends on the particular characteristics of these type relationships. In section 4.5, we introduced two particular notions with respect to these type relationships that are of interest to the request processor's operation. These notions are the characterisation of type relationships as either simple or composite, and their characterisation as either runtime or development time ones. As we have already seen above, the former characterisation introduces the distinction between simple and composite offers, where the latter require the support of the composition facility. As a result, the request processor needs to distinguish and to treat differently these two kinds of offers, composite ones need to be forwarded to the composition facility before they can be returned to the requester. The latter characterisation of the type relationships introduces two distinct query interfaces that clients can use, a runtime one and a development time one. The two query interfaces serve a different purpose and consequently consider type relationships differently during the query expansion phase.

More specifically, the purpose of the runtime query interface, as its name implies, is to be used by a software system during its operation, i.e. at runtime, to discover and locate appropriate components that can seamlessly be integrated into the running system. As a result, in this case the request processor should only provide offers that are directly usable, i.e. offers that provide quite strong guarantees of interface conformance. This kind of guarantees is exactly what the characterisation of type relationship as runtime ensures. Consequently, when a client uses the runtime query interface to submit a request, the requestor processor during the query expansion phase should only consider runtime type relationships. Note that the characterisations of type relationships as either runtime or development time, and as simple or composite are orthogonal. This means that a type relationship can be characterised as a runtime and simple, or a runtime and composite one. In the latter case, the composition facility knows how to produce directly usable offers out of the matched ones. However, this production process may require that offers of a number of different component types are available. Unless all of them are available, we result with incomplete compositions, which are not directly usable and thus should not be returned to the client.
In contrast, the purpose of the development time query interface is to be used by tools controlled by software developers to explore what kind of components and component types are available. As a result, in this case the trader is not restricted to provide immediately usable offers. This means that the request processor can use both runtime and development time relationships in the query expansion phase. Moreover, in addition to complete composite matches, the request processor may also provide incomplete ones, in order to indicate to the developer the missing parts. Furthermore, to show the kind of component types it is aware of it could also provide the list of the ones used in the query expansion phase, even if no offers were available for them.

In the discussion so far of the two query interfaces, we assumed that in both cases client requests are expressed with respect to the name of the component type, offers of which are required. In fact, in the case of the runtime query interface this has to be the case as it is the only way to guarantee that the trading process is fully automated. However, this is not the case for the development time query interface, as it is not bound by the same requirement. In order to make this point clearer we need to examine in a bit more detail how the query expansion phase of the trading process is carried out. For this purpose the request processor needs to have access to the various type relationships maintained by the type manager. As depicted in Figure 5.8 the type manager is responsible for constructing and maintaining a graph that captures all the established type relationships. In this graph, nodes represent the various types known to the type manager and edges represent established relationships between these types. Such a graph can be constructed with the support of a service like the CORBA relationships service [56]. Then, when client requests are expressed in terms of the name of a component type, the query expansion phase involves locating the node for the requested type on the graph, and starting from this node traverse all the edges representing type relationships of particular kinds as determined by the request. Every time an edge is traversed a new conformant type is discovered and offers of this type are also considered. The traversal continues until either we have exhausted all conformant types, i.e. followed all the paths of type relationships of the particular kind to the end, or we reached certain limits imposed by the request itself. Note that this process is similar to the one that service traders currently follow when they traverse their interface subtype hierarchy. It should be
clear that this process can be fully automated and that it can be implemented using quite
efficient graph traversal algorithms. This is why we require the request processor to
follow this process when the runtime query interface is used.

However, in the case when the development time query interface is used we can
also allow alternative ways of expressing requests. A particularly interesting alternative is
to provide template-based querying. In this case the client instead of requesting offers of
a particular component type it provides a template component type specification and
requires any offers that may match the template. As a result, we remove the requirement
for the client to know the name of the type that satisfies its requirements. In template-
based querying the request processor will have to first introduce the template type
description provided as a query type to the trader’s type manager. Note that query types
are similar to the virtual types in [39], in the sense that the trader’s offer manager is not
aware of these types and as a result no offers of them will ever be available. The
introduction will result in the type manager establishing all possible type relationships
between the query type and other types, i.e. placing the query type into the type
relationship graph. When this process is completed then the request processor can go on
to carry out the query expansion following the same process described above. The
problem with template-based querying is that as we are going to see later on, the
introduction of the query type into the type relationship graph cannot in general be fully
automated (see section 5.2.3). Figure 5.9 provides a more detailed architecture for the
request processor.

In addition to the name of the component type or the component type template,
query interfaces also allow the client to control the way in which the request processor
carries the trading process through the use of type and offer selection criteria, and offer
preference rules. More specifically, the type selection criteria allow clients to control the
query expansion phase by either restricting the type relationship edges that are followed
by the graph traversal algorithm, or by requiring particular component type properties to
be defined. In the former case the restrictions may exclude composite type relationships,
as compositions in general introduce some computational overheads (see section 5.2.4 for
more details). Or, they may exclude any type relationships that do not define
equivalences. Or, they may even place a lower limit to the strength of type relationships
considered. Note that when we refer to the strength of the type relationships, we refer to the strength of the predicate used to relate their behavioural descriptions (see the lattice in Figure 4.5). For this purpose we consider domain enhanced name-based type relationships as incomparable to the semantics-based ones. In contrast, the offer selection criteria do not affect the query expansion phase, but the way the offer manager selects offers for the considered component types. The selection is based either on the range of property values supported by the offers or by the kind of evidence supporting the offer’s conformance to a component type, or the particular attributes of the offer. The preference rules determine how the request processor should order the matched offers before returning them to the client. Following a similar approach to the CORBA trading service [56], we can define a number of different ordering strategies, like random, first, or stronger in terms of either the strength of the relationship or the strength of the conformance evidence, or even based on the values of particular properties, etc. In addition to the pre-defined strategies we could also allow clients to provide their own ordering function.

![Request processor architecture](image)

**Figure 5.9. Request processor architecture.**

Finally as we mentioned in section 3.2, traders may be linked to form federations. When the trader takes part in a federation, the request processor as part of its role needs also to forward queries to the federation. What exactly this involves depends on the type of federation.
5.2.2. Offer manager

The role of the offer manager is on one hand to maintain the set of available component offers, i.e. enabling component providers to export new offer and remove previously exported offers, and on the other hand to find any available offers for the particular component types provided by the request processor. In order to carry out its role the offer manager requires the support of a component offer repository. This repository includes all the currently available offers organised by component type. This means that the offer manager cannot accept any offers for component types that it is not aware. Moreover, it also needs to be notified every time the type manager introduces a new component type.

Component offers comprise of the following elements:

- **Component Home Reference.** As we saw in section 3.1 contrary to services, components require certain configuration to be carried out before they can be used. At the least, this configuration requires that they are connected to services providing the functionality the component requires, i.e. they are connected to the kind of services their required service types describe. More often than not, their extra-functional properties would also need to be configured to reflect the characteristics of the particular deployment environment. This means that for most extra-functional component properties instead of a single property value offers would have a range of values within which the property can be configured. Note that service trading has been criticised for its inability to support this kind of configuration of the extra-functional properties of services [222]. As a result, state-of-the art component platforms like EJBs or CCM provide component homes, which support a number of component types and are responsible for managing the lifecycle of their components including the configuration. Accordingly, clients receive references to the component homes that they can use to create appropriately configured components.

- **Component Type.** This is the component type that the reference claims to conform to. The reason we are talking about claims of conformance is because of the fact that although we can easily establish the conformance of an offer to
a particular component type with respect to its syntactic features, doing so with respect to the specifications of behaviour is quite challenging and in the general case cannot be fully automated. As a result, the offer manager just accepts the exporters’ claims for the component type of their offers and classifies them according to this type. Finally, note that for the syntactic features of the component type conformance establishment relies on the introspection features of the component platform.

- **Conformance Evidence.** Since, the offer manager cannot establish the conformance of offers to their type, following an approach similar to the one advocated in [223], it requires that exporters provide some evidence supporting their claim. The evidence can take a variety of different forms, from formal proofs that a proof checker can automatically verify, to rigorous proofs that developers can verify, to sets of test cases that component passes, or even developer testimonies of successful uses of the particular component. It should be clear that different kinds of evidence provide different assurances of component conformance. As a result, the trader allows clients to specify which types of conformance evidence are acceptable through the use of offer selection criteria. In general, the issue of managing conformance evidence requires a comprehensive framework for reasoning about the validity and quality of the provided evidence, which is outside the scope of this thesis.

- **Offer attributes.** These express characteristics of the particular component offer that may also be taken into account as part of the offer selection criteria. For example, they may provide information about the developer, the usage pricing, the licensing, etc of the component. A particularly interesting use of offer attributes would be to support some form of leasing as supported in Jini (see section 3.1.1).

A more detailed architecture of the offer manager is provided in Figure 5.10. Based on the above characteristics of the offers, the architecture comprises of two repositories: the component offer repository, which stores the available offers and supports offer selection, and the conformance evidence repository, which stores the
conformance evidence for the available offers and supports the reasoning about validity and quality of the provided evidence.

Figure 5.10. Offer manager architecture.

5.2.3. Type manager

Figure 5.11. Type manager architecture.
As we mentioned in section 3.2 type managers provide two functions: (a) type description, which provides operations to describe and compare types, and (b) type management, which provides operations to record type names and their relationships. In order to support these two functions the trader’s type manager comprises of the following components (see Figure 5.11):

- A type model repository, which is responsible for storing the descriptions of all the types and the specifications of all the type relationships known to the type manager. At the same time, as some of the type descriptions may refer to concepts and types standardised within particular type definition domains, the type model repository may also be associated to a number of domain repositories, one for each of the type definition domains known to the type manager. Note that the domain repositories can either be part of the type model repository or separate repositories to which the type model repository has access. The type manager uses the type model repository and the domain repositories, whenever a new type is introduced in order to establish any relationships the type may have to other known types. We should also point out that query types produced in the case of template-based querying, are also introduced to the type model repository. In this case, though, the type description is in fact the provided template. Moreover, note that although in general may be useful to remove types or type relationships from the repository we do not really consider this issue any further. Furthermore, so far although we have listed the various elements of the description of each type of the component type model, together with rigorous definitions of alternative ways of description (see section 4.4), we have not fully defined a particular language for this purpose. We believe that because of its extensible nature XML is the best candidate for this purpose. In fact, we could see the various type description elements in section 4.4 as elements of an appropriate XML schema with named types and imported type definition domains representing links between XML documents and external references respectively. Note also that, as we mentioned before, our choice of UML/OCL for the description of the behavioural specifications of the types, means that these behavioural
specifications can also be expressed in XML using XMI [34]. In this case, the
type model repository is in fact an XML document repository.

- A type directory, which is responsible for recording the names of all the types
  known to the type manager. Any of the currently available directory services
  mentioned in section 3.1.2 can be used for this purpose. The type manager
  uses the type directory first to enforce the uniqueness of type names according
  to the rules of the type naming scheme described in section 4.3, and second to
  locate the requested component type in the type relationship graph during the
  query expansion phase. Consequently, the type directory stores for each type
  name a reference to the corresponding node of the type relationship graph.
  Note that the type directory also includes entries for the query types.

- A type relationship graph manager, which is responsible for constructing and
  maintaining the graph that records all known relationships between types. The
  maintenance of the graph is primarily concerned with the introduction of new
  types and the establishment of their relationships to the existing ones. As these
  relationships also involve semantics-based ones that are expressed in terms of
  predicates over the behavioural specifications of the types, this process
  requires the support of a theorem prover and as a result cannot be fully
  automated. Note that this is the reason behind our decision to not support
  template-based querying as part of the runtime query interface. Moreover, the
  type relationships graph manager also provides the interface that the request
  processor uses to implement its graph traversal algorithm. In fact, it provides
  two such interfaces, one for the runtime querying interface and one for the
  development time query interface. Each of these interfaces provides a different
  view of the type relationship graph. It is interesting to note that we can further
  increase the usefulness of the trader in the development process by exposing
  the graph manager's interface for development time querying, and allowing
  developers to define their own traversal algorithms. Using this approach we
  could define for example a traversal algorithm that takes a set of service types
  and finds a set of component types that provide as many of these service types
as possible while at the same time minimises service type overlaps and gaps, like the one described in [192].

The type relationship graph is the focus of our attention in the rest of this section, as it plays a central role in the trading process. In particular, we examine how the type manager handles the introduction of new types to the graph.

**The type relationship graph**

The type relationship graph comprises of nodes that represent types and edges that represent type compatibility/conformance relationships. There are a number of different kinds of nodes, each representing a different kind of types of the component type model. As type compatibility/conformance relationships are primarily defined between property types, action and action interface types, service and component interface types, service and component behavioural interface types, and action and component types, these are the only kinds of nodes that the graph considers. All these kinds of nodes are characterised as simple, as they represent a single type of the component type model. In addition to simple nodes, the graph also includes composite nodes, too. Composite nodes, as the name indicates represent a component type, which results from the composition of a set of component types. These nodes are introduced as a way to support the extension of the trading process from one-to-one to one-to-many matches. During the trading process composite nodes result in the request processor submitting a number of requests for component offers to the offer manager, one for each of the component types comprising the composite node. For this reason, in this case we refer to composite offers. Note however, that the term composite offer is also used to refer to the offers of conformant component types according to composite type compatibility/conformance relationships. Both simple and composite nodes include besides the name of the type they represent also a reference to the type description store in the type model repository. In addition to this, composite nodes also include a composition strategy that the composition facility can use to compose its constituent component types.

Similarly to nodes, edges are also of different kinds, each kind representing a particular type compatibility/conformance relationship. Moreover, the edges are also divided into runtime and development time ones, according to the characterisation of the type relationships they represent. Note that the runtime view of the graph includes only
runtime edges, while the development time view includes all edges. Furthermore, edges are also divided into simple and composite ones. The main difference between the two is that the latter, similarly to composite nodes, also include a composition strategy. In addition to the type compatibility/conformance edges described above, the graph also includes inclusion edges. These edges are used to represent the inclusion relationships between different kinds of types, e.g. the fact that a service type includes a number of action types, or a component type includes a number of service types, etc. Inclusion edges are used in order to establish relationships between types, in terms of their comprising types, i.e. establish service type conformance in terms of the conformance of their respective types. Note that as we mentioned above during the query expansion phase of the trading process, only edges representing semantics-based or domain-enhanced name-based compatibility/conformance types are considered. This ensures the semantic nature of the trading process.

In order to demonstrate what the construction of the graph involves let us consider how new component types are introduced. First, we should point out that in the beginning the graph includes all universal property types, as well as all the standardised types of the type definition domains that the type manager is associated to. In general, the component type introduction process involves two phases: a decomposition and a type relationships establishment phase.

More specifically, in the decomposition phase, all the constituent types of component type are identified, i.e. component property types, component behavioural interface types, component interface type, service types, etc. down to the level of action interface types. At the same time, nodes for all these types, and inclusion edges between the corresponding nodes are created. Note that if some of the produced types were already included in the graph, no new nodes are created for them.

In the type relationships establishment phase we examine all the produced types of the decomposition phase in order to establish their type compatibility/conformance relationships to the other types known to the type manager. For each established relationship the appropriate edge between the related types is introduced in the graph. This process exploits the inclusion edges created in the previous phase and establishes type relationships in a bottom up manner, starting from property types and action-related
types, then moving on to service-related types and finally to component-related types. At the same time, in order to make the process more manageable, we use the type relationships with the lowest computational cost first to reduce the scope of types that need to be examined for compatibility/conformance. So, we start with name-based relationships that are quite easy to establish with the help of the type directory. We then move on to domain-enhanced name-based relationships. In parallel, we also start the examination of structure-based relationships that can also easily discard large numbers of unrelated types. Note that if both a name-based and structure-based edge exists between two types, then a syntactic-based edge can also be introduced between them.

As we move to semantics-based relationships, the computational cost rises dramatically. For this reason, we do not attempt to establish such relationships unless a name- or structure-based relationship already holds between the two types under consideration. The idea is to check whether syntactic similarities between the types are an indication of a stronger relationship. Before we can check for any predicates over the behavioural specifications, we need to identify abstraction or simulation functions, or relations between the respective service and component behavioural models. This is the first point where the developer introducing the new component type could provide some help. After these functions or relations are identified, we can go ahead to check the predicates between the behavioural specifications. This involves proving the truth of logical expressions and requires the support of a theorem prover. In this process we can also utilise any available information of the application domain specific domain, i.e. common behavioural attributes, common lemmas, etc. Again in this point the help of the developer may also be required. Note that as domain-enhanced name-based relationships reflect semantic content, the related types do not have to be examined for semantics-based relationships.

The final issue that we need to be careful about during the establishment of type compatibility/conformance relationships is that the process is quite often recursive and particular care need to be taken to avoid blocking the process due to cycles. For this purpose we suggest that every time the establishment of a particular relationship is predicated upon establishing a certain relationship between other types, we introduce a provisional edge predicated on a certain obligation. As we move up in the type
relationships establishment process, some obligations will be satisfied, in which case the corresponding provisional edges can be turned into proper ones, other obligations may be refuted, in which case the provisional edges are removed, and other obligations will remain. For the obligations remaining at the end of each level we construct obligation dependency graphs and perform cycle detection on them. Any detected cycle can be safely removed and the dependent obligations comprising can be considered satisfied. This may lead to further provisional edges being replace by proper ones.

In conclusion, the construction of the type relationship graph is a very challenging process, and can only be manageable by following the above incremental process.

5.2.4. Composition facility

As we mentioned above, the composition facility is the component introduced to the trader architecture in order to take advantage of composite types and type relationships. This kind of relationships produce composite offers, i.e. ones that require some processing before they are directly usable by clients. So, the role of the composition facility is to carry out the processing of the composite offers. There are multiple levels of processing that may be required, starting from adaptations of the component interface, going on to protocol mediation, or full-scale composition. In either case, there are two ways in which this kind of processing may be accomplished, statically, i.e. using pre-defined components encapsulating the required processing, or dynamically, i.e. using some form of automated construction of the components encapsulating the required processing. Moreover, in both cases the composition process involves the construction of a component home able to construct an interception hull within which the components of the simple offers comprising the composition will be placed. The idea behind the interception hull, as the name implies, is that it intercepts any client calls, carries out the necessary processing, and then forwards them to the internal components. Any response from the internal components is also intercepted by the hull, processed and then forwarded back to the clients. Note that this interception hulls are in fact simplified versions of the component containers that state-of-art component platforms like EJBs and CCM support.
Consequently, in order to guide the composition facility, composite type relationships need to be accompanied by a composition strategy. The strategy is expressed as a script in the sense describe in [227]. This script is executed by the composition facility, and specifies how the interceptor hull should be constructed. Moreover, in the case of static composition the script will also identify the particular pre-defined components to be used from a repository of available such components. Figure 5.12 provides a more detailed architecture for the composition facility.

![Composition Facility Architecture](image)

**Figure 5.12. Composition facility architecture.**

In general, the full specification of a scripting language for composition strategies is outside the scope of this thesis. However, we should point out that there is already significant work that could provide a good starting point for any such attempt. More specifically, with respect to interface adaptation Konstantas in [224] offers a full language for expressing object interface mappings, which we believe can be extended to cover the case of components too. In the case of protocol mediation Yellin and Strom in [200] describe a language based on abstract state machines for protocol specification, which can also be used for the automated construction of mediators. Furthermore, the work on the specification of connectors in software architecture [225, 226] can also provide the basis for extending the above approaches to full fledged composition. Finally, we should note, that state-of-the-art component platforms already provide certain features that can form the basis for the automated construction of the interceptor hulls. The most notable example is the case of CORBA with its support for the Dynamic Skeleton
Interface (DSI). DSI allows servers to dynamically interpret received requests and decide which particular method call should be used to process them. As a result, code that performs the required adaptations can be inserted as part of this process. Realising the potential of call interception techniques to enhance the functionality of pre-existing server implementations, CORBA has gone ahead to provide a specification for portable interceptors as part of its overall architecture [24].

5.2.5. The role of a SECT-enabled trader in enterprise information system development

In this section we briefly show how a SECT-enabled trader can be used in the development of enterprise information systems like the ones that provided the motivation for this thesis (see section 1.1). These systems are usually the result of the integration of a number of existing enterprise systems. We assume that each of the existing enterprise systems comprises of a number of components and is supported by a SECT-enabled trader (the dashed clouds in Figure 5.13). In order to develop the new enterprise system (the outside cloud) the existing systems need to first federate their traders. This federation creates a virtual trader that is able to trade components from both of the existing systems (the dashed trader on the right). The federation requires that the two existing traders merge their type managers. This requires that bridges be established between their known types. These bridges are either provided by their shared application domain specific domains, or are directly constructed by establishing type compatibility/conformance relationships between types of the two type managers. In this case, of particular interest may also be cross-domain name-based type relationships, which are similar to the domain-enhanced type relationships we defined in section 4.7. Their main difference is that instead of requiring equality of the type names defined using terms from the vocabulary of a single domain, we require that the type names are defined using different but semantically related terms from the vocabulary of different domains. Another way of directly constructing bridges is through the introduction to either type manager's type relationship graph of composite nodes comprising of a mix of types from the two type managers. Having established the bridges between the types of the two type managers, we can then extend the query expansion phase of the trading process to cross over the
type manager boundaries. Finally, the request processor can then query the corresponding offer managers for matching component offers.

![Diagram of Enterprise Information System Development](image)

**Figure 5.13. Enterprise information system development.**

### 5.2.6. Concluding remarks

In this section, we presented an architecture for a SECT-enabled trader. Taking advantage of the particular characteristics of the SECT component type model the trader is able to bring the notion of service trading into the domain of component-based development. At the same it also has the potential to support higher quality trading. The quality improvement stems from considering only semantics-based type conformance/compatibility relationships during the query expansion phase of trading. As a result, the matching offers of component types conformant to those requested are more likely to be relevant, leading to an improvement in the precision of the trading process. The quality improvement also stems from the introduction of composite type relationships, and composite nodes in the type relationship graph. These, supported by the composition facility, can be used in the query expansion phase of trading to allow matching of semantically related offers despite their structural differences, leading to an improvement in the recall of the trading process. It is interesting to note that this improvement in recall does not compromise precision. Moreover, the introduction of composite nodes in the type relationship graph and the composition facility also provides
a good foundation for the extension of trading from a one-to-one into a one-to-many matching process.

Nevertheless, it is important to point out that the above improvements are predicated on: (a) the establishment of semantics-based type relationships in the type relationship graph, and (b) the establishment of the conformance of component offers to their claimed semantic types. As both of these processes cannot be fully automated, the runtime use of a SECT-enabled trader is of limited value. As a result, the effectiveness of SECT in a highly dynamic context is also limited. In this context, a wealth of information incorporated into application domain specific type definition domains can seriously improve the utility of SECT. Accordingly, SECT can only deliver the dynamic enterprise collaborations envisioned in section 1.1 in cases where bridges between the application domains of the collaborators, such as those discussed in section 5.2.5, have already been established.

At the same time, we need to draw attention to the fact that even if SECT is limited to development time use, the realisation of the above-presented architecture involves some serious technical challenges. These challenges are primarily in supporting the establishment of the crucial semantics-base type compatibility/ conformance relationships and the conformance of provided offers to their component type. With respect to the former challenge the main difficulty lies in formally grounding UML/OCL to the foundation of a theorem prover that can subsequently be used to aid the establishment of the semantics-based type relationships. Although, there is evidence that this is possible [231], there is still some way to go before the details are fully worked out. With respect to the latter challenge, the requirement for offer exporters to make available conformance evidence provides a good foundation. However, a framework for evaluating the likelihood of offer conformance under the light of the available evidence needs to be constructed. Once again although in principle the construction of such a framework is possible, further research is required to realise it.

5.3. Summary

In summary this chapter evaluates the SECT component type model with respect to our claim that SECT can be the solution to the initial problem statement of this thesis,
specifically to bring techniques for service location and discovery in networked and distributed systems into the domain of component-based development, and in doing so to improve the trading process. We carry out our evaluation by providing a series of example type definitions according to the SECT component type models and an architecture for a SECT-enabled trader. The former demonstrate the expressiveness of the component type model in specifying the characteristics of components and their behaviour in particular. The latter provides some useful insights into how a trader can be developed that supports component-based development and is able to improve the quality of the trading process.
Chapter 6. Conclusions and future work

In this chapter we conclude this thesis with a discussion of its original contributions; its relevance in the light of recent development in enterprise information systems; the main advantages and disadvantages of our overall approach; and potential directions of future work. More specifically, in section 6.1, we identify the main contributions of this thesis and we demonstrate how they enhance the state-of-the-art. In section 6.2, we examine the main contributions of the thesis from a web services perspective. In section 6.3, we identify the main advantages and disadvantages of our approach and summarise our main conclusions. Finally, in section 6.4, we identify direction for future work, with respect to both further enhancing our component type model and applying it in areas beyond enterprise information systems.

6.1. Contribution and originality

The primary contribution of this thesis is the introduction of the notion of Semantically Enhanced Component Trading (SECT) that is solidly founded on a component type model incorporating notions of semantic types and type relationships. In this context, the distinctive characteristic of the described work is its unique mix of ideas from the areas of component retrieval in software reuse, object interoperability, and software architecture to produce a coherent framework within which the type model is defined. In this process particular attention has been paid to critical evaluation of the underlying concepts from a component software point of view. Furthermore, this work also explains why because of its particularly rich set of type compatibility/conformance relationships, SECT enables higher quality trading than current service trading approaches by increasing both the recall and the precision of the trading process.

More specifically, in order to clarify further the original contributions of this thesis, we directly compare it to work that in our opinion is the most closely related. Indulska et al. in [1] present a very comprehensive type model for service trading. Although we follow the same approach for the definition of our component type model, the fact that our type model is designed from a component software perspective means that it has certain distinct characteristics, like component related types and type
relationships. Moreover, although they advocate the extension of service trading type models to include semantic notions of types and type compatibility/conformance it does not really describe how this can be introduced. In contrast our component type model describes very particular ways in which this can be achieved.

Christiansen et al. in [4] take type management approach to the reuse of services. Similarly to this thesis, they also recognise the potential of signature and specification matching in the context of type management. However, although they describe in detail how signature matching can be incorporated into type management, in the case of specification matching their discussion is fairly superficial. In this thesis both kinds of matching are examined with the same rigour and are fully incorporated in the type model. Furthermore, this thesis extends their work by applying signature and specification matching to component types.

The work that is most closely related to this thesis is the work on the COTS Trader [202], a trader particularly designed for COTS components. To our knowledge this is the only other work that examines service trading from a component software perspective. As a result it has a number of similarities to this thesis. First, it also incorporates semantic notions of types and type relationships in the trading process. Second, it also views components as sets of provided and required services, while enabling the trader’s matching process to handle them. Third, it also places certain emphasis on the description of extra-functional property of the traded components. Fourth, it also acknowledges the need for more flexibility in the matching process and the need for one-to-many matches in addition to the one-to-one ones. Because of these similarities it may appear that this work invalidates this thesis. However, a closer look reveals that this is not the case.

More specifically, although it introduces semantic notions of types that capture both the protocol and semantic-level interoperability aspects, the emphasis is certainly on the former. Consequently, the latter receives very little attention beside the mention of exploiting the work on specification matching and behavioural subtyping. As a result, in contrast to this thesis, the issues of how to construct component behavioural models and how to combine the approaches to specification matching and behavioural subtyping into a coherent framework for type compatibility/conformance are not addressed. Moreover,
as we have argued before, although protocol level interoperability is crucial in guaranteeing the correct use of software components, it is not a major concern when searching for them as quite often incompatibilities can be masked out with the use of appropriate mediators. Furthermore, although it introduces components providing and requiring a set of service in the trading process, it does not introduce the notion of component types. As a result, the complications arising from the interaction between the various services of a component are not taken into consideration. This is not the case in this thesis as component constraints have been introduced as part of component types for exactly this purpose. Besides, with respect to the description of the extra-functional properties, in this thesis we incorporate the notion of extra-functional contract types that set a frame within which particular aspects can be specified, and we also introduce the notion of extra-functional domains allowing for contract types to be specified independently from particular components. Additionally, we do not only recognise the need for flexible and one-to-many matches, but we also provide detailed descriptions of the former and the notion of composite matches for the latter.

We should though point out that because of the emphasis on COTS components the work on COTS Trader pays particular attention to the description of the commercial properties of components and does not require behavioural descriptions to be in a particular specification language. However, it does not really address the issue of how the differences in the specification languages are handled during the trading process. At the same time, by considering the trader as an Internet service, it also pays particular attention to the issues of offer discovery supporting both active and passive discovery. Finally, it shows how the COTS Trader could be incorporated into a component-based development process.

As we mentioned before, another strand of similar work, are approaches that enhance service trading or discovery protocols in general by incorporate knowledge management techniques, like semantic networks [170] and concept ontologies [172, 175] (e.g. [44, 155, 156, 171, 174]). We should also point out that the work on agent matchmaking is also following this particular strand [203], in the sense that agent capabilities are described with respect to an ontology. Then, using this ontology together with reasoning techniques from description logics [204], matchmaking agents acting in a
similar way to traders can discover agent with specific capabilities. This work is similar in the sense that it is driven by the same motivation that introduced the notion of SECT, i.e. the limited effectiveness of syntactic-based discovery. However, we consider this work as complementary to this thesis. In fact, as we mentioned in section 4.7, our notion of application specific domains can be in fact extended to include a domain ontology, instead of the domain vocabulary. In this case, our notion of domain enhanced type relationships can also be extended to incorporate concept subsumption into the predicates relating the names of types and their features.

Finally, we should point out that to our knowledge we are the first to integrate the various function specification matches that appear in the literature into a single construct (see Figure 4.4).

6.2. Relevance in the era of web services

In the previous section we discussed the original contributions of this thesis and we showed how it enhances on the state-of-the-art. Before we continue to summarise our conclusions, in this section we turn our attention to web services and we briefly examine the contributions of this thesis from their perspective. The main reason behind our decision to include this examination is the growing interest within both academia and industry on web services in recent years, combined with claims that web services are the next step in the evolution of software development (see [206] for example), in the same sense as component software was the step after object-oriented software. If these claims were true, they could seriously undermine the relevance of the work in this thesis. We believe that this is certainly a matter that requires further investigation.

The first question that we need to answer is what are web services. They are a number of XML-based standards that enable the discovery and use of Internet enabled services [205]. These standards include: (i) the SOAP protocol, for XML-based communication between services, (ii) the Web Service Description Language (WSDL), for XML-based description of service operations and to bind to them, and (iii) the Universal Description, Discovery and Integration (UDDI) specification, for directory-based discovery of services described in WSDL. The main motivation behind the introduction of web services was to exploit the popularity of established web standards in
the area of enterprise systems integration. According to the web services view, developers would consider enterprise systems as collections of interconnected Internet enabled services, which can be reconfigured and reconnected to accommodate changes in the enterprise environment. This view gives rise to the notion of service-oriented architectures according to which enterprise systems are organised.

The web services view presented above looks quite similar to the component-based view we described in section 1.1. Consequently, the second question that we need to answer is in which way web services are different to the services that components offer. According to Szyperski [207] services, in the context of service-oriented architectures, are the same as the services that components offer with the important difference that they are already deployed in a particular computational environment and are supported by an administrative authority. As a result, they can provide guarantees about the quality of their service and they can take part in service-level agreements. Neither of these are possible for the services of components, as the quality of the services they provide can only be guaranteed with respect to particular characteristics of the computational environment, while service-level agreements require authorities as signatories of the agreement. The main consequence of this difference is that while services in the context of service-oriented architectures can describe their performance related aspects in a fixed way, services, in the context of component software have to describe them in a parameterised way.

The above discussion suggests that despite the hype that surrounds them web services are not really that different. As a result, we could try to apply SECT in service-oriented architectures. In which case, the third question that we need to answer is how can this be done taking into consideration the particular technical characteristics of web services. In answering this question first we need to point out that all the component platforms that we examined in section 2.3 already provide some support for web services. Consequently, we can rely on these platforms to provide the underlying mechanisms necessary for locating particular services. In fact, service location information is explicitly provided in WSDL descriptions as part of the service bindings. In general, a web service may define a number of different bindings, each encoding location information in a different manner. Second, in web services the discovery problem is
addressed by UDDI. In fact, the UDDI repository plays exactly the same role as a trader
within a service-oriented architecture. UDDI suffers from the same problems as service
trading, since WSDL descriptions do not include explicit specifications of the service
behaviour. So, to introduce SECT in service-oriented architectures we need to first extend
WSDL descriptions to include specifications of service behaviour. In fact as WSDL
descriptions are in XML we can introduce this extension in a way that does not affect
SECT-unaware UDDI repositories and services. Second, we need to introduce SECT-
enabled UDDI repositories in order to take advantage of the behavioural specifications
during the discovery of services. Our proposed trader architecture could provide the basis
of these SECT-enabled UDDI repositories. We believe that instantiating the trader
architecture in the context of web services would be relatively straightforward.

In closing, we need to note that there is already work attempting to address the
problems that the syntactic nature of WSDL and UDDI poses. This work combines the
semantic web technologies and web services to produce semantic web services [208].
The approach is based on the introduction of a new language for the description of
services DAML-S [209], which uses ontologies to describe the semantics of service
operations. Moreover, this work also applies techniques from agent matchmaking to
extend UDDI to support semantic matches between web services [210]. Although, it is
intended for this work to be extended to also incorporate specification matching,
currently the approach is exclusively based on semantic distance between concepts of the
service ontology. Consequently, as we mentioned in the previous section this kind of
work is in some complementary to our approach. However, a particular interesting aspect
of this work is that it also explores ways in which semantic matching can be used to drive
service composition [211].

In conclusion, the emergence of web services does not invalidate SECT. In fact,
SECT could provide the same benefits in the area of service-oriented architectures.
However, it would be even more beneficial if the SECT is combined with the work in
semantic web services into a unified framework.
6.3. Conclusions

Current enterprise information systems need to exhibit high degrees of flexibility and dynamism. The only way to provide such flexibility and dynamism is by developing them as collections of components that can be dynamically reconfigured and recomposed into new configurations. This continuous reconfiguration process requires the support of a service that is able to select and locate the appropriate components. In order for any such service to be developed, it is necessary that we have a clear idea of what a software component is and what are its defining characteristics. Definition 2.1 is selected as the most appropriate for this purpose.

At the same time, an analysis of various solutions to the problem of service location and discovery in networked and open distributed systems from a component software perspective shows that:

- Component references and component home interfaces, as supported by state-of-art component platforms, are an adequate solution to the location problem.
- Service trading is the most appropriate candidate as a basis for a component location and selection service. However, its syntactic nature is a serious problem.

Semantically Enhanced Component Trading (SECT) is a new kind of trading that has the potential to overcome the problems of service trading by introducing a component type model that incorporates semantic notions of types and type compatibility/conformance relationships. The type model requires that type definitions include explicit descriptions of type behaviour, which can be taken into consideration when defining type compatibility/conformance relationships. In addition to this, the most interesting characteristics of the type model are the following:

- Component types comprising of sets of provided and required services.
- Action type as abstractions of both events and operations.
- Behavioural and syntactic interface types for action, services and components.
- Action behavioural models including model programs.
- Service behavioural models including invariants, history constraints and protocol-level interoperability constraints.
• Behavioural models for the provided services of components including `MODEL_VARIABLE` types.
• Specification of extra-functional properties of services and components with respect to extra-functional contract types.
• A type naming scheme that accommodates the standardisation of types with the support of conformant type names.
• Rigorous definition of type descriptions that support the definition of type relationships.
• Name-based, structure-based, syntactic-based and semantics-based notions of type compatibility/conformance relationships, incorporating signature and specification matching.
• Development time and runtime, as well as composite type relationships.
• A template for the rigorous definition of type relationships.
• Support of both parametric and inclusion polymorphism.
• Use of inheritance as a mechanism for supporting incremental definition of types.
• Type definition domains that support the definition independently of any particular types of both extra-functional properties, and of standardisation of application domain specific concepts and types.

A number of examples demonstrate that the type model allows for expressive type specifications that are able to precisely capture both syntactic and semantic variations between types. Moreover, a trader architecture has been designed that utilises the type model to enable higher quality trading compared to current service traders. The quality improvement is both in terms of precision and recall. The improvement in precision results from the use of semantics-based type compatibility/conformance relationships, while the improvement in recall from the use of composite type relationships and the support of a composition facility. Other interesting features of the architecture include:

• The support for both development time and runtime trading utilising the respective kinds of types.
• The support of template-based querying during development time trading.
The association of conformance evidence with offers that is used in the evaluation process of the selected offers.

With respect to the realisation of the provided trader architecture the main challenge is in supporting as far as possible the inference of semantics-based type relationships. In the general case the inference process requires the support of theorem prover and as a result it cannot be fully automated. Furthermore, UML with each formal component OCL is still at the early stages of formalisation. At the same time, the implementation of a composition facility that goes beyond massaging of component interfaces is also a serious challenge.

6.4. Directions for future work

There are a number of directions in which this work can be extended in the future. We classify the directions in three categories: (a) extensions of the type model, (b) realisation of the trading architecture, and (c) refinement of the type model for particular domains. Within the first category we envisage the following extensions:

• **Incorporation of ontologies in type definition domains.** As we discussed in section 4.7 the application domain specific domains of our component type model currently include a domain vocabulary. This vocabulary is utilised in the definition of domain-enhanced type compatibility/conformance relationships. We could enhance the component type model by replacing the domain vocabulary by a domain ontology. Having a domain ontology available we can also extend our notion of domain-enhanced type relationships to incorporate concept subsumption, i.e. ontological type relationships. Furthermore, the domain ontology can also be used to define notions of conceptual distance between types related by ontological relationships. As establishing ontological relationships is in general less computationally demanding than establishing semantics-based relationships, when introducing new types we could use the conceptual distance to limit the scope of types that we need to consider for semantics-based conformance.

• **Introduction of more elaborate composite type relationships.** As we discussed in section 4.5 composite relationships are those that are not
immediately usable, as they require some form of adaptation to the interface of the matched types. In the component type model we currently only consider relatively simple composite types, i.e. those resulting from the transformation signature matches or the specification matches that are not reuse ensuring. In general, the mechanism of composite types can be used to define any kind of one-to-many type relationships. As a result, a direction for future work would involve to identify useful such relationships and to investigate how they can be established by a type manager in an automated way. However, particular care would have to be taken in order for these relationships to not have a serious effect on the overall performance of the process of introducing new types.

- **Extension of the behavioural models to cover concurrency.** As we discussed in section 4.2.5, service behavioural models are expressed in terms of pre-/post-conditions, invariants, history constraints and model programs. However, it is known these are not sufficient to capture aspect relating to concurrent behaviour of actions. In order to address this deficiency rely/guarantee clauses have been proposed. As a result, another enhancement of the type model would include the introduction of these clauses and an investigation of how they can be incorporated in semantics-based type relationships.

- **Formalisation of extra-functional aspects.** As we discussed in section 4.2.4, extra-functional aspect of types are described using properties, i.e. name-value pairs. Although, this approach is currently the norm it suffers from the lack of precise semantics for the property types. The use of extra-functional contract types introduces some implicit semantic information, but does not fully resolve the problem. At the same time, in the literature there are attempts to formalise extra-functional aspect, e.g. [212] attempts to do so in the area of transactional properties. As a result, a possible enhancement of the type model would involve introducing such formalisation in the definition of extra-functional properties and to investigate how these formalisations could be incorporated in the definition of semantics-based type relationships.
The second category involves work towards the realisation of the provided trader architecture. In this respect, the most challenging component of the architecture is the type manager. The main challenge in the development of the type manager lies in the process of inferring semantics-based type relationships. As we mentioned in our conclusions, this process requires the support of a theorem prover. However, currently UML/OCL for the behavioural descriptions is not fully formalised for UML/OCL expressions to be amenable to theorem proving. Addressing this challenge is another direction of feature work.

Finally, the third category refers to work that attempts to capture the semantics of a particular application domain through the definition of specific kinds of component types and type relationships that in some sense refine the generic types and types relationships of the component type model.

6.4.1. Applying SECT to context-aware systems

In this section we attempt to illustrate further the third category of directions for future work by exploring briefly how SECT can be used for the processing of contextual information.

A system is considered context-aware if it is able to perceive the context within it operates and can adapt its behaviour in response to changes in this context [213]. A typical example, of context-awareness is a location-aware printing service, which prints documents to the printer closest to the user's current location. Context-awareness is considered as an integral part of ubiquitous computing [214]. Capturing users' context and managing contextual information is currently a "hot" research area. The main challenge arises from the fact that capturing users' context usually involves gathering low-level sensor data, which are then combined and transformed to higher-level context information. A component-based approach seems particularly suited for this kind of processing. Such an approach would involve representing the various sensors and sensor data transformers by software components that are combined to produce the required kind of contextual information. An important observation is that the same kind of contextual information can usually be produced through a variety of sensor data and transformations. Moreover, as ubiquitous systems are highly dynamic the set of available
context processing components is constantly changing. In order for context-aware applications to cope with these changes, they require the support of a service that is able to discover and locate appropriate context processing components. Such a service seems very similar to the component location and selection service we described in section 1.1.2. As a result, such a service could benefit from the use of SECT.

In order to deploy SECT in this context we need to use its component type model to describe both the types of the various context-processing elements and the various types of context. Then, based on these types we can define the various context transformation processes as composite type relationships. A SECT-enabled context trader can then use these relationships to find sets of context-processing elements, which realise alternative context transformation processes for the same type of context. It is interesting to note that the types of common context-processing elements as well type relationships representing common context transformation processes can be standardised within a context-processing domain and be made available to all context traders. A more detailed presentation of the ideas sketched above can be found in [228].
Bibliography


10. Sheth and V. Kashyap, So Far (Schematically) yet So Near (Semantically), in Proceedings of the IFIP WG 2.6 Database Semantics Conference on Interoperable
44. A. Cox, An Exploration of the Application of Software Reuse Techniques to the Location of Services in a Distributed Computing Environment, Master Thesis Department of Computer Science Trinity College, September 1994.


70. E.S. Dictor, Visual Basic Controls in a Nutshell, O’Reilly, July 1999.
83. O. Nierstrasz and L. Dami, Component-Oriented Software Technology, in [55], pp. 3-28.
Rio de Janeiro, Brazil, October 2001, available online at http://www.cs.ucl.ac.uk/staff/C.Alves/SBES'01_Alves.pdf.


114. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns CD: Elements of Reusable Object-Oriented Software, Addison Wesley, 1998.


147. O. Nierstrasz, Regular Types for Active Objects, in [55].


