CONTENT-ORIENTED ARCHITECTURE FOR CONSUMER-TO-BUSINESS eCOMMERCE

By

JOSHUA HARRY GREENBERG

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY UNIVERSITY OF FLORIDA 2002
Dedicated To
My Loving Parents and My Loving Wife
ACKNOWLEDGMENTS

I express my sincere gratitude to my advisor, Dr. Douglas D. Dankel II, for his steadfast guidance and sincere advocacy during the course of this work. I wish to thank Dr. Stanley Su, Dr. Joseph N. Wilson, Dr. Manuel E. Bermúdez and Dr. Suleyman Tufekci for serving on my supervisory committee and for their careful review of this dissertation. I also wish to thank Dr. Abdelsalam (Sumi) Helal for his contribution.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xi</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xiv</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Consumer-to-Business eCommerce</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Understanding the Problem</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Outline</td>
<td>8</td>
</tr>
<tr>
<td>2 CONSUMER-TO-BUSINESS</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Ontology and Linguistics</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1 Categorizing Reality</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 Models for Categorization</td>
<td>13</td>
</tr>
<tr>
<td>2.1.3 Computer Science Perspective on Ontology</td>
<td>15</td>
</tr>
<tr>
<td>2.1.4 Problem of Sharing</td>
<td>21</td>
</tr>
<tr>
<td>2.1.5 Linguistics</td>
<td>23</td>
</tr>
<tr>
<td>2.1.6 Agent Communication Languages</td>
<td>29</td>
</tr>
<tr>
<td>2.1.6.1 Description of the FIPA-ACL</td>
<td>33</td>
</tr>
<tr>
<td>2.1.6.2 KQML</td>
<td>44</td>
</tr>
<tr>
<td>2.2 Programming Language Design</td>
<td>49</td>
</tr>
<tr>
<td>2.3 Persistence Mechanisms</td>
<td>62</td>
</tr>
<tr>
<td>3 THE PROBLEMS OF C2B</td>
<td>67</td>
</tr>
<tr>
<td>3.1 Research Goals</td>
<td>70</td>
</tr>
<tr>
<td>3.1.1 Mobile Shopper</td>
<td>70</td>
</tr>
<tr>
<td>3.1.2 Emergency Room</td>
<td>71</td>
</tr>
<tr>
<td>3.1.3 Job Searching</td>
<td>71</td>
</tr>
<tr>
<td>3.1.4 Insuring The Family</td>
<td>72</td>
</tr>
<tr>
<td>3.1.5 Proactive Treatment</td>
<td>72</td>
</tr>
<tr>
<td>3.1.6 Large Scale of Software Design</td>
<td>73</td>
</tr>
<tr>
<td>3.1.7 Outline of Goals</td>
<td>73</td>
</tr>
<tr>
<td>3.2 Additional Details</td>
<td>74</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Tabular presentation of RDF data</td>
<td>19</td>
</tr>
<tr>
<td>2-2</td>
<td>KQML performatives</td>
<td>46</td>
</tr>
<tr>
<td>4-1</td>
<td>SSA-CommunicationIntent</td>
<td>95</td>
</tr>
<tr>
<td>4-2</td>
<td>SSA-CommunicationChannel</td>
<td>96</td>
</tr>
<tr>
<td>4-3</td>
<td>SSA-CommunicationMode</td>
<td>96</td>
</tr>
<tr>
<td>4-4</td>
<td>SSA-CommunicationDependence</td>
<td>97</td>
</tr>
<tr>
<td>4-5</td>
<td>SSA-CommunicationReliability</td>
<td>98</td>
</tr>
<tr>
<td>4-6</td>
<td>SSA-CommunicationSemantics</td>
<td>98</td>
</tr>
<tr>
<td>4-7</td>
<td>SSA-ReplicationLevel</td>
<td>99</td>
</tr>
<tr>
<td>4-8</td>
<td>SSA-ReplicationSemantics &lt;ACCESSSTYPE P&gt;</td>
<td>100</td>
</tr>
<tr>
<td>4-9</td>
<td>SSA-TransactionSupport</td>
<td>102</td>
</tr>
<tr>
<td>4-10</td>
<td>SSA-Transaction Semantics</td>
<td>103</td>
</tr>
<tr>
<td>4-11</td>
<td>SSA-ServiceDiscovery</td>
<td>104</td>
</tr>
<tr>
<td>4-12</td>
<td>SSA-Execution Deployments</td>
<td>105</td>
</tr>
<tr>
<td>4-13</td>
<td>SSA-PersistenceFormat</td>
<td>106</td>
</tr>
<tr>
<td>4-14</td>
<td>SSA-Presentation</td>
<td>106</td>
</tr>
<tr>
<td>4-15</td>
<td>SSA-Connectivity</td>
<td>107</td>
</tr>
<tr>
<td>4-16</td>
<td>SSA-User Types</td>
<td>108</td>
</tr>
<tr>
<td>4-17</td>
<td>SSA-Maintenance Schedule</td>
<td>109</td>
</tr>
</tbody>
</table>
4-18  SSA-Supported Modifications .................................................................................. 114

7-1  Methods of the UDDI interfaces .................................................................................. 226
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Example ECML form</td>
<td>3</td>
</tr>
<tr>
<td>1-2</td>
<td>Agent-to-agent interoperation stack</td>
<td>6</td>
</tr>
<tr>
<td>2-1</td>
<td>Graphical presentation of RDF data</td>
<td>20</td>
</tr>
<tr>
<td>2-2</td>
<td>Model of FIPA request protocol</td>
<td>40</td>
</tr>
<tr>
<td>2-3</td>
<td>Model of FIPA contract-net protocol</td>
<td>42</td>
</tr>
<tr>
<td>5-1</td>
<td>Hierarchical content model</td>
<td>145</td>
</tr>
<tr>
<td>5-2</td>
<td>Edge-labeled hierarchical content model</td>
<td>146</td>
</tr>
<tr>
<td>5-3</td>
<td>Edge-labeled model enables expression of the car ownership relation</td>
<td>148</td>
</tr>
<tr>
<td>5-4</td>
<td>Acquisition date is not a true property of car</td>
<td>148</td>
</tr>
<tr>
<td>5-5</td>
<td>If relationships are on edges, then they do not belong as node types</td>
<td>149</td>
</tr>
<tr>
<td>5-6</td>
<td>Adding multiple edges between nodes</td>
<td>150</td>
</tr>
<tr>
<td>5-7</td>
<td>Models for recursive relationships</td>
<td>152</td>
</tr>
<tr>
<td>5-8</td>
<td>Reference nodes as a substitute for cycles</td>
<td>155</td>
</tr>
<tr>
<td>5-9</td>
<td>An evolution of models demonstrating cycles, reference nodes, and separation of concepts from relationships</td>
<td>157</td>
</tr>
<tr>
<td>5-10</td>
<td>Relational schema</td>
<td>160</td>
</tr>
<tr>
<td>5-11</td>
<td>Three types of content sharing</td>
<td>169</td>
</tr>
<tr>
<td>5-12</td>
<td>Separating implementation, interface, and content sharing</td>
<td>170</td>
</tr>
<tr>
<td>5-13</td>
<td>Perspective domain graph</td>
<td>172</td>
</tr>
<tr>
<td>5-14</td>
<td>Car-ownership PDG</td>
<td>174</td>
</tr>
</tbody>
</table>
7-12  Factual information pellet providing details about a car for sale.......................... 256
7-13  Using a PDG to request content.................................................................................. 259
CONTENT-ORIENTED ARCHITECTURE FOR CONSUMER-TO-BUSINESS eCOMMERCE

By

Joshua Harry Greenberg

August 2002

Chair: Douglas D. Dankel, II
Department: Computer and Information Science and Engineering

Consumer-to-Business (C2B) systems represent the future of eCommerce. Using natural language as a basis, and remaining keenly aware of its potential pitfalls, we describe a new communication model based on a content-biased language (CBL). It is shown that the requirements of a C2B system cannot be satisfied with anything less than the stretchability of a CBL. Based on that realization, the remainder of the work strives to present a usable representation for a CBL, as well as an architecture for using that representation. The result of this work is the description of a new software quality measure called stretchability, as well as the introduction of perspective domain graphs (PDGs), external open ontological type systems (EOOTS), and global and constituent systems. The dissertation closes with the definition of a new distributed system design called the Content-Oriented Architecture (COA).
CHAPTER 1
INTRODUCTION

Imagine a world where information is at your fingertips. Imagine further that the information is timely and relevant. The promise of the Internet lies hidden beneath a morass of incoherent web pages and undiscerning search engines; but that is only a symptom of the problem. The cancer is manifested in the generality. The information you need, the information to which you ascribe greatest import, has one glaring characteristic. It is the information concerning you, relating to your own individuality. But that information is sadly nowhere to be found.

I submit that such an omission is no longer acceptable. The individual on the Internet, whether consumer, producer, or otherwise, is the center of his own universe and all other entities should revolve around him. I do not want to see the same view of the Internet as you, and I certainly do not want my preferences to be dictated by a statistic. We are all unique individuals with personal attributes and specific intentions, and we want an Internet responsive to our needs.

Envision a new world where the technologies of the Internet, wireless devices, and semantically self-adapting software combine to make each user feel that he is the only user, where the individual can target the market not as a passive statistic but as an active participant. Begin to expect direct marketing, but begin to embrace the concept of marketing of yourself. At the push of a button, we will tell the world what we want it to know, and with an almost magical flicker, the world will reply to our bidding.
1.1 Consumer-to-Business eCommerce

Given its seemingly endless media attention, the term Business-to-Business (B2B) is familiar to most people as designating the Internet-based supply-chain oriented transactions executed between corporations. Business-to-Consumer (B2C) has also entered the business executive’s arsenal of Internet-based acronyms, and it generally signifies the set of activities surrounding the marketing and selling of goods by companies to individuals. To distinguish online auctions and other noncorporate business activities, one may apply the term Consumer-to-Consumer (C2C). While all of these buzzwords designate important and profitable computing paradigms, it is a less frequently discussed model that may prove most revolutionary. Brown [16] and Livraghi [78] define Consumer-to-Business (C2B) as the comparison shopping activities performed on-line by a user before purchasing a product. While this definition may accurately represent current implementations, it barely scratches the surface of what is possible. By enabling direct-marketing and self-marketing, the C2B concepts proposed and clarified in this thesis allow consumers to do far more than simply compare prices and characteristics. They will place consumers on an equal footing with corporations.

If we move away from the notion of C2B as comparison-shopping, then at present, the most representative implementations are generally categorized as wallet software systems. Until very recently, the most formalized attempt at wallet software systems was the Electronic Commerce Modeling Language (ECML). ECML allows "consumers to enter personal details once into the wallet software, which could be called up as needed to make payments to retailers" [101:19]. Once the information has been entered, order forms for Internet transactions can be filled automatically with data such as billing preferences, shipping information, identity, credit-card numbers, and digital certificates
The list of companies backing the ECML standard includes American Express, MasterCard, Visa, International Business Machines Corp., Compaq Computer Corp., Dell Computer Corp., Microsoft Corp., Sun Microsystems Inc., Cybercash, and America Online. An example ECML form appears in Figure 1-1 and is based on the standard form presented by ECML.org [40].

![Example ECML form](image)

James McQuivey, an analyst with Forrester Research, states, “what escaped wallet software vendors is that the average online user simply did not know how to download and install software” [103]. This quote underscores a number of very important points. First is the question of whether wallet software should maintain the user’s data on his personal PC or a central server? We feel the answer lies in a hybrid approach accommodating both privacy and flexibility requirements. The next important point is that wallet software is viewed as a small, downloadable application that has no use
outside of on-line purchasing. We strongly disagree with this view and hope to show that
wallet software is as important an application as the desktop word-processor. Certainly,
one would state that Microsoft Word was poorly designed since users cannot easily
download and install it off the Internet. Finally, we feel the term wallet may be a weak
metaphor for our proposed system. Perhaps vault is more appropriate.

In an interesting turn of events, as this thesis nears completion, Microsoft has
announced a new wallet-based technology, code named Hailstorm. According to the
latest press releases, the Hailstorm product will be given the name .NET MyServices, and
the wallet portion will be referred to as a safe-deposit box. While no complete version of
Hailstorm is currently available, these naming choices, and the very existence of the
product, demonstrate consensus concerning the evolution of C2B from comparison-
shopping services to a complex consumer-based set of applications.

This expanded view of the wallet more closely matches the domain of electronic
commerce, which as described by Honeyman [61], "involves everything one can do in the
physical world: advertising, shopping, bartering, negotiating contracts and prices, bidding
for contracts, ordering, billing, payment, settlement, accounting, loans, bonding, escrow,
etc."

While we propose a new architecture for C2B systems, this thesis also strives to
clarify the underlying issues involved in creating such a system. Consequently, these
issues span a vast number of areas including topics from disciplines such as computer
science, linguistics, and philosophy. Future system designers would be wise to keep
these issues in focus, and future implementations should be judged on how well each of
the issues is handled.
1.2 Understanding the Problem

Before venturing into the text of this dissertation is it useful to establish a clearer picture of the problem. In the context of computer science research, Consumer-to-Business (C2B) issues have not received much attention. For that reason, a primary goal, and likewise, a primary contribution, of this work is to help define the topic and detail its requirements. First and foremost, C2B seeks to make consumer information an asset to the consumer, and at the same time seeks to ensure that external demands for that information do not place unnecessary burden on the consumer. As it turns out, these are complimentary requirements.

Consider an individual shopping for a new insurance policy. This consumer would like a new policy, identical to his existing policy, but with a reduced deductible. In his search for a new provider, the consumer may choose to use the Internet. Unfortunately, the web sites provided by each insurer have custom interfaces, and the consumer is forced to retype the identical information at each location. Furthermore, before this process can even begin, he must read through his current policy description and extract the important details. One simple solution to this problem is for all insurers to get together and create a single, standardized form. However, this simple solution is not quite so simple, and furthermore it does not scale to enable sharing between car insurers, medical insurers, medical doctors, pharmacists, and others.

A better solution to the insurance shopper’s dilemma relies on the existence of a C2B infrastructure. First, the consumer needs an easily accessible, easily updateable repository that stores his personal information. This information needs to be protected from unauthorized access, but just as importantly it needs to be semantically meaningful to those parties that are granted access. Second, the consumer needs a way of sending
this information to another consumer, a corporation, or any other interested party. Similarly, it would be useful if the consumer could simply place this information in an externally accessible location as a means of advertising himself. Finally, the recipient of the consumer’s information must be able to understand the content of what has been received, and must also be able to request additional information if what was supplied was insufficient or incomplete.

At this point four primary goals have been identified:

1. Maintaining personal information,
2. Transmitting or advertising personal information,
3. Ensuring information is understood by a receiver, and
4. Enabling the receiver to request additional information.

As expected, these goals are equally as valuable for B2B as they are for C2B. However, a solution for the C2B problem must not be restricted to a single business domain, must be easy to use, must be ultimately extensible, and finally must be inexpensive, or free for the consumer. In other words, a consumer must be able to store

<table>
<thead>
<tr>
<th>Content Biased Language (CBL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTR and Matchmakers (replacing UDDI, WSDL)</td>
</tr>
<tr>
<td>Endpoint Access Protocol (SOAP)</td>
</tr>
<tr>
<td>eXtensible Markup Language (XML)</td>
</tr>
<tr>
<td>Internet Protocols (HTTP, TCP/IP)</td>
</tr>
</tbody>
</table>

Figure 1-2. Agent-to-agent interoperation stack
details about any relevant concept, at any level of granularity, and be able to send those details as meaningful content to any number of receivers. Furthermore, the receivers
must be able to call back to the consumer and request additional information. These callbacks should introduce minimum burden to the consumer. That is, the additional requests should be answered automatically if the data is available in the consumer’s data repository. Otherwise, the consumer should be presented with a human-friendly form through email, on a PDA, or in any other fashion such that the additional details can be entered into the form, stored into the repository for any future requests, and sent to the requester to help finish the transaction.

The technologies required to realize some of the requirements presented above are just beginning to emerge into the realm of mainstream computing. However, one very fundamental issue still remains. Specifically, there is still no mechanism defined to enable independent consumers, corporations, and other organizations to define the details of the information they wish to communicate, and to establish, in a computer and human accessible fashion, the semantics of that information. In addition to a number of other contributions surrounding the domain of C2B, this dissertation presents a Content-Biased Language (CBL) that acts as the final layer on the agent-to-agent interoperation stack illustrated in Figure 1-2. We explain why such a language is required, uncover the requirements of the language, and ultimately define a representation for the language and explain how it is used in a high-level construct called the Global System.

This research makes the following contributions to address the problems discussed above:

1. Clarification of the Consumer-to-Business research problem,
2. Introduction of stretchability as a software quality measure for distributed systems,
3. Introduction of six faceted requirements (6FR), software specification axioms (SSAs) and requirements patterns (RPs) to help formalize the requirements for stretchable systems,

4. Definition of a content-biased language (CBL) as a means to enable stretchability using content-biased communication between autonomous agents,

5. Description of external open ontological type specifications (EOOTS) and the global type repository (GTR) as the basis for realizing a content-biased language,

6. Specification of perspective domain graphs (PDGs) and their XML-based representation as the content description language (CDL) and content description language schema (CDL-Schema),

7. Introduction of an architecture based on the new notion of a Global System composed of constituent systems and situated atop a global framework comprised of the GTR, EOOTS, and a number of other components, and

8. Construction of a new system design paradigm succinctly described as the content oriented architecture (COA).

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 outlines related research from many different fields including ontology, linguistics, agent communication languages, inter-process communication, type system design, dispatch mechanisms, and semi-structured data. In Chapter 3, the goals of a C2B system are examined and a number of representative use-cases are discussed. Chapter 4 sets the stage for the remainder of the dissertation by introducing a software quality measure called stretchability and by presenting the concepts of global and constituent systems. An exploration of the features of stretchable software provides a set of requirements for a C2B system and also introduces the important notion of content-transfer. In Chapter 5, the form and meaning of content is thoroughly examined and ultimately leads to the introduction of content-biased languages. Chapter 6 takes the first steps towards design of a C2B system by examining a number of important components that will appear in the
global system and its associated constituent systems. Finally, in Chapter 7, all of the work is brought together under the umbrella of a new system design paradigm called the Content Oriented Architecture. Chapter 8 concludes the dissertation and suggests areas for future study.
CHAPTER 2
CONSUMER-TO-BUSINESS

This chapter serves two primary purposes. First, it clearly identifies the many research areas related to problems in the C2B domain. Second, the exposition serves as a didactic aid for any particular areas not immediately familiar to the reader. Because the material presented here precedes any detailed discussion of C2B, no sincere attempt is made to correlate the information with specific problems. Any required correlations are made explicit in the following chapters.

2.1 Ontology and Linguistics

There are two very important questions to be asked in the context of communication. First, what is the underlying structure used to represent both tangible and abstract concepts from our domain of discourse? And, second, how do we convert that representation into descriptive and effective communications, and furthermore, how might we understand or quantify the efficacy of those communications? Neither question lends itself to simple solution. And, as we shall see, there is a rich body of research dedicated to understanding these problems.

2.1.1 Categorizing Reality

Over two millennia ago, Aristotle composed *Categories* and *Metaphysics*, in which he tried to answer the first question posed above. To do so, Aristotle divided the world (universe) into substances and accidents. A substance exists on its own, has well-defined boundaries, properly contains no other substances, exists continuously through time, takes up space, and always maintains a unique identity. Accidents may be contrasted
with substances but also exist because of them; “they are two distinct orders of being. The former [substances] *endure* self-identically through time; the latter [accidents] *occur*: they unfold themselves through time, and are never present in full at any given instant during which they exist” [113]. A person may be considered a substance, though his attached head may not. However, upon decapitation, the head may be considered a substance. An accident may be considered a process, event, quality, or condition that is borne by a substance. In the context of a person, accidents include his knowledge, his feelings, his relationship to his possessions, or the act of his decapitation. For future explication, it is also useful to introduce the notion of a collective, which is essentially a collection of substances, possibly volatile.

A very interesting categorization of accidents is discussed by Smith [113]; the categories follow:

1. Non-relational accident – associated with only a single substance,
2. Relational accident – associated with multiple substances and thus somehow joins the substances,
3. Comparitives – a comparison of some particular attributes of two substances (i.e. is-longer-than),
4. Cambridge Relations – relation imposed by societal reflection (i.e., father-of) and,
5. Instituted accident – an accident of a collective that may persist even if particular substances of that collective are removed.

Given the notion of a substance, one may logically ask where a substance resides or similarly, what defines the boundaries of a substance. Aristotle considered this topic in his Physics and posited that a substance has a place that is distinct from its state though separable from its state. A place has size and shape but no composition. Furthermore, the substance enveloped by the place is encased exactly at the joint of their boundaries,
though the place exists independently of the particular substance. As such, the substance
is replaceable, and the replacement is said to occupy the same place.

While it is useful to discuss the boundaries between substances, it is also interesting to
consider the boundaries within. As stated by Smith, “each substance is marked by the
possibility of partition along an indefinite number of interior lines of division” [113].

Such lines of division may be genuine boundaries or fiat boundaries. A genuine
boundary is identified by some qualitative heterogeneity of the substance itself (e.g., a
wheel on a car), whereas a fiat boundary does “not respect qualitative differentiations or
spatio-temporal discontinuities in the underlying territory” [113] (e.g., the northern
hemisphere of the globe).

The motivation for studying boundaries is often to provide reasonable points for the
division of reality. Defining such points can assist in the creation of a formal ontology as
envisioned by Husserl in his *Logical Investigations*. In the realm of computer science,
and especially artificial intelligence, a formal ontology is often simply referred to as an
ontology and is defined as an explicit formal specification of the terms in a given domain
and their interrelationships [56]. This contrasts with the more widely accepted definition
of ontology as “the science that deals with the nature and organization of
reality”[112:287]. An important sub discipline of ontology, called mereology, focuses on
the notions of part versus whole. When mereology is combined with topology, “the study
of geometrical properties and spatial relations unaffected by the continuous change of
shape or size of figures” [89], the result is a powerful field called mereotopology. Smith
said mereotopology allows “the formulation of ontological laws pertaining to the
boundaries of interiors of wholes, to relations of contact and connectedness, to the
concepts of surface, point, neighborhood, and so on” [112:288].

2.1.2 Models for Categorization

Smith [113] distinguished between two views of the world. The set-theoretic view
approaches the world as starting from a set of primitive atoms. These atoms may then be
composed to create higher-level objects whose properties we wish to investigate. An
alternative view is motivated by the work of the ecological psychologist, Roger Barker,
in the context of physical-behavioral units. A physical-behavioral unit is composed of
psychological and non-psychological entities and also includes their interactions. The
unit “occupies a determinate, bounded locale having observable geographical, physical
and temporal attributes and having boundaries which are coincident with the boundaries
of the behavior that takes place within it” [113]. Thus an example of a physical-
behavioral unit is the business transaction that occurs when an individual seeks to
purchase insurance. The unit includes all the people and objects and all their interactions.
One final important aspect of a physical-behavioral unit is that certain parts can be
modified – for instance, the particular insurance agent can be replaced without distorting
the essence of the unit.

Though not the explicit motivation of his paper, an interesting approach to modeling a
physical-behavioral unit may be found in Smith’s work on quantum mereotopology
[114]. Quantum mereotopology extends the standard mereological topology by
introducing a theory of granular partitions that permits discussion of “objects (at given
resolutions) without at the same time talking about all the parts of those objects (at all
finer resolutions)” [114:26]. This ability is important since “cognition induces a certain
sort of quantization (or granularization) on objects in space and time” [114:26]. The
theory is set up to produce better results when clarifying the relationships between an 
element and its set. These relationships [114] include the following:

1. The relation between an object and its location.
2. The relation between an object and a concept under which it falls.
3. The relation between an object and a kind or category to which it belongs.
4. The relations an object may bear to intervals on quantitative and qualitative 
scales.
5. The relation between an object and its role or function or office or niche.
6. The relation between an object and the corresponding entry in a list or record in a 
database.

A granular partition is used to divide the world into cells, where each cell may or may 
not contain a corresponding object. A periodic table is cited as an example of a granular 
partition over the matter of the universe. As in all granular partitions, some details are 
made explicit, while others are not considered. The primary advantage realized in using 
granular partitions is that it restricts the fusion of parts not only in space, but also in time 
and under the changes that may occur through time. The cells in a granular partition 
extend Aristotle’s notion of place in that an object is located entirely in a cell but is not 
necessarily contained exactly therein.

It is important to understand that partitions are not composed of the objects contained 
in their cells. Furthermore, cells are not dependent on the particular objects they contain. 
A partition is not an object itself, but is instead a division of reality at some level of 
granularity. The selected divisions induce cells, and at any given time, certain objects are 
contained in those cells. Thus, a cell may at different times contain different objects, or 
may always contain the same object regardless of how it changes over time – the decision 
depends on the particulars of the granular partition. Thus, a partition with a cell labeled
Albert Einstein will always point to the well-known physicist, but a partition with a cell labeled car parked in space 47 will point to whatever car happens to occupy that space.

The theory of granular partitions is extended over the time domain through the use of histories. An illustrative example is to consider a partition with cells labeled Dr. Jones’ graduate students. One option would be to design a granular partition containing one-cell that simply contains all such students – this cell would be constant over all time. Another alternative is to create a partition that immediately divides the domain such that there is one cell for each student. However, consider that the set of students will change over time. To accommodate this situation, the notion of histories, or sequences of granular partitions, is introduced. For the graduate student example, a new granular partition may be constructed at the end of each semester. Then, the sequence of such partitions would represent the history of students advised by Dr. Jones. Just as the granularity of cells in a partition is selected, so is the granularity of time in a history. In the given example, the index times were chosen as the close of each semester. A more formal treatment of these concepts, including a set of operations over cells, partitions, and histories, is provided by Smith and Brogaard [114], and the reader is referred there for any additional details.

2.1.3 Computer Science Perspective on Ontology

According to Gruber, an ontology is a “specification of a conceptualization” [55:200]. A conceptualization is, roughly, a set of concepts and the relationships that exist between them. The concepts and relationships chosen are those that have some particular importance in a domain of agent interaction. What is perhaps more important than the ontology itself, is the notion of ontological commitment defined as “an agreement to use a vocabulary (i.e., ask queries and make assertions) in a way that is consistent (but not complete) with respect to the theory specified by an ontology” [55:202].
A definition that resides closer to a possible implementation is provided by Noy and McGuinness [94:3] who state, “an ontology is a formal explicit description of concepts in a domain of discourse (classes), properties of each concept describing various features and attributes of the concept (slots), and restrictions on slots (facets).” The same paper goes further to describe a knowledge base as a set of individual instances of the classes in an ontology. To better understand the motivation for such a definition, it is useful to examine a number of existing implementations.

The work of Corcho and Gomez-Perez [25] compares numerous different ontology specification languages based on criteria including “readability (how things are said), expressiveness (what can be said) and inference (what can be obtained from the information represented)” [25:1]. The three criteria are considered in the context of the five major ontological components identified by Gruber [55]:

1. Concept – any topic about which one wishes to reason.
2. Relation – a statement regarding a set of concepts such that particular instances of the concepts make the statement either true or false.
3. Function – an operation over a set of concepts that returns another concept.
4. Axiom – a representation of a statement that is always true.
5. Instance – a particular value in the domain of a concept.

A concept is generally described through a set of slots or attributes. Similar to class attributes in object-oriented languages, these slots may be static, static const, instance, or polymorph. Furthermore, slots usually have restrictions placed on their domains through the use of facets. Facets can provide a type (domain) for a slot but may also provide a default value, cardinality constraint, and others.
Functions and relations serve to encode the majority of information in an ontology, and often a concept may even be modeled as a unary relation. Overloading of the relation component does not stop there, and in a number of ontology specification languages (e.g., Ontolingua, OKBC, LOOM) there is no syntactic distinction drawn between functions or relations. In other languages such as CycL [28], functions are differentiated from relations (called predicates in CycL) in that the former return a CycL term as a result. For example, consider the following function definition:

\[
(\text{arity GovernmentFn 1})
\]

\[
(\text{arg1 isa GovernmentFn GeopoliticalEntity})
\]

\[
(\text{result Isa GovernmentFn RegionalGovernment}) [28]
\]

It is reasonably clear that the expressions define a function called GovernmentFn that takes one argument of type GeopoliticalEntity, and returns a value of type RegionalGovernment. This differs from a predicate such as the following:

\[
(\text{isa residesInDwelling BinaryPredicate})
\]

\[
(\text{arg1 isa residesInDwelling Animal})
\]

\[
(\text{arg2 isa residesInDwelling ShelterConstruction}) [28]
\]

In this case, the statements define a binary relation called residesInDwelling. The relation takes two arguments, an Animal and a ShelterConstruction, and returns true if the instances provided validate the predicate; otherwise, it returns false.

The expressions, or CycL formulas, shown above, are representative of the general syntax of the CycL language. CycL is a declarative language based on first-order predicate calculus and includes representations for terms (such as semantic constants, non-atomic terms, variables, numbers, strings, etc.). These terms can be combined to create CycL sentences (formulas with no free variables), which are then aggregated to
form a knowledge base. The meaning of any constant (user-supplied string name) in a CycL knowledge base is completely determined by the set of sentences involving that constant.

Another declarative language based on first-order predicate calculus is the Knowledge Interchange Format (KIF). KIF is “formal language for the interchange of knowledge among disparate computer programs” [46]. KIF has declarative semantics, is logically comprehensive, supports the representation of meta-knowledge, and provides for expression of non-monotonic reasoning rules. Similar to CycL, the knowledge of KIF can be broken down into four major expression types: terms, sentences, rules, and definitions. “Terms are used to denote objects in the world being described; sentences are used to express facts about the world; rules are used to express legal steps of inference; and definitions are used to define constants” [46]. For the purposes of ontology construction, Gruber extended KIF with a frame-language to permit the more familiar and accessible notions of classes, slots, facets, and the subclass relation. The result is the Frame Ontology language called Ontolingua [55].

One option for extracting data from Ontolingua, or any other frame-based knowledge resource, is Open Knowledge Base Connectivity (OKBC) [18]. OKBC includes commands for querying and manipulating the information stored in a knowledge representation system (KRS) using generic operations on frames, slots, and facets. Once again, the familiar notions of objects and subclassing are borrowed from object-oriented languages.

Leaving the realm of KIF, or first-order predicate calculus based languages, one finds the Resource Description Framework (RDF) [74]. RDF is a frame based knowledge
representation system based on an eXtensible Markup Language (XML) syntax. It differs from KIF and CycL in that no reasoning mechanism is provided. The goals of RDF include resource discovery, cataloging, knowledge-sharing between agents, content rating, and a number of other web-related issues. The RDF data model [74] “defines a simple model for describing interrelationships among resources in terms of named properties and values … however, [it] provides no mechanisms for declaring these properties, nor does it provide any mechanisms for defining relationships between these properties and other resources. That is the role of RDF Schema” [13].

The RDF Schema data model elevates the property (or slot) to the position normally filled by classes (concepts). That is, the world is viewed from the perspective of properties. Associated with each property there are zero or more domain properties and an optional range property. The domain specifies which classes the property may apply to, and the range determines the type of result expected when applying the property. For instance, consider the sentence “Charles Dickens is the author of the book Oliver Twist.” In the subject, predicate, object breakdown suggested by RDF, the components of the sentence would appear as shown in Table 2-1.

Table 2-1. Tabular presentation of RDF data

<table>
<thead>
<tr>
<th>Subject (resource)</th>
<th><a href="http://www.books.com/OliverTwist">http://www.books.com/OliverTwist</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate (property)</td>
<td>Author</td>
</tr>
<tr>
<td>Object (literal)</td>
<td>“Charles Dickens”</td>
</tr>
</tbody>
</table>

RDF also proposes the use of “partially labeled directed graph[s] in which nodes are either unlabeled – these are also called anonymous or blank nodes – or else labeled with either URIs or literals; arcs are labeled with URIs; and distinct labeled nodes have different labels” [60]. For the example above, the graph is illustrated in Figure 2-1.
In the language of RDF Schema, the predicate, Author, has domain Book, and a value in the range of string literals. In the particular case above, the book is uniquely identified by the URI “http://www.books.com/Oliver Twist” and the author is specified as “Charles Dickens.” A corresponding RDF Schema may appear as follows.

```
<!-- File URI: http://www.examples.com/books# -->
<rdf:RDF xml:lang="en"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <rdfs:Class rdf:ID="Book">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
  </rdfs:Class>

  <rdf:Property ID="Author">
    <rdfs:domain rdf:resource="#Book"/>
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
  </rdf:Property>
</rdf:RDF>
```

The corresponding RDF/XML document for the particular book would then take the following form.

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:books="http://www.examples.com/books#"/>
  <rdf:Description about="http://www.books.com/Oliver Twist">
    <books:Author>Charles Dickens</books:Author>
  </rdf:Description>
</rdf:RDF>
```
2.1.4 Problem of Sharing

Creation of an ontology for use of a single program would almost be counterproductive. The work required to produce an ontology is difficult and time-consuming, and the primary benefits are obtained only when multiple parties realize a new ability to communicate about common tasks. Unfortunately, the involvement of multiple parties into almost any endeavor acts to hinder its progression. This reality also holds true for collaborative ontology construction, and research has been undertaken to help alleviate some of the major obstacles.

The Ontolingua Server, an extension, or application on top of the KIF based Ontolingua language, strives to help achieve “consensus on common shared ontologies by geographically distributed groups” [42:1]. Farquhar, Fikes, and Rice [42] describe the Ontolingua Server as providing the following features:

1. A semi-formal representation language that supports the description of terms both in natural language and formally in a rigorous computer interpretable knowledge representation language.

2. Browsing and retrieval of ontologies from repositories.

3. Assembly, customization, and extension of ontologies from repositories.

4. Facilities for translating ontologies from repositories into typical application environments.

5. Facilities for programmatic access to ontologies so that remote applications have reliable access to up-to-date term definitions.

6. Support for distributed, collaborative development of consensus ontologies. [42:2]

There are really two distinct, but related, requirements that arise in this context. First, what support structures can be created to assist geographically separated groups in
construction of a communally acceptable ontology. And, second, how can these structures be extended to assist ontology development by autonomous and temporally independent groups. The latter problem is somewhat more difficult since the parties involved may have conflicting requirements but have no means of discussing acceptable accommodations. Furthermore, any changes made after the initial design must consider the impact on current dependent products – about which we may know nothing. Of course, one may argue to the contrary and claim that by eliminating the potential for communication, we increase the potential for solution. This seems a somewhat cynical view, but it may be quite accurate. People tend to be rather inflexible concerning their preferred dialect or vocabulary and may simply refuse any accommodations. Uschold [130] categorizes these issues to help identify the requirements that arise in trying to standardize the usage of terminology throughout an organization. The requirements cited are as follows:

1. Local autonomy – local groups need to be able to own, create, maintain and most importantly, use their own terminology.
2. Flexibility and ease of maintenance – it must be relatively easy to make changes to the local terminology.
3. Global access – local groups need access to things that are classified under terms from different groups.
4. Stability – behavior of systems using the ontologies must remain relatively stable. [130:2]

The notion of one, single, universally accepted global ontology suitable for all domains is something of a Holy Grail for philosophers and computer scientists alike. And like the Grail, it remains undiscovered and unlikely. A more reasonable option may be the creation of numerous, domain specific local ontologies. Whether or not these local ontologies should converge to some global source remains an issue of current debate.
However, it is generally accepted that some mapping mechanism to bridge the concepts between local ontologies is useful, if not mandatory. Even in the context of simple markup languages such as XML, work has been undertaken to enable conversion from one document format to another (e.g., XSLT). The problems that arise concerning unique naming of concepts, cyclical dependencies, creation of logical inconsistencies, and many others are still areas of active research.

2.1.5 Linguistics

At this point, we shift our focus onto the second question posed above, namely, given some chosen ontological representation of reality, how do we construct descriptive and effective communications, and furthermore, how might we understand or quantify the efficacy of those communications? To help answer this pair of questions, it is convenient to appeal to a common decomposition used in the study of language. In general, any tool for communication (spoken language, sign language, written language, etc.) can be analyzed from three distinguishable perspectives. Syntax formalizes the placement and relationship of lexemes. Semantics, usually associated with meaning, may be more aptly described as the relationship between the lexemes and what they represent with respect to some universe of discourse. And, finally, pragmatics may be said to account for meanings that semantics ignores [75]. A useful interpretation of this statement is to include in pragmatics the study of how communication affects the receiver (hearer) over and above whatever information may have been explicitly conveyed. The complementary notions of informative intent and communicative intent support this view. The former is the actual explicit meaning of an utterance, while the latter is the meaning intended by the speaker.
Since pragmatics is tightly connected to the effect an utterance has on its hearer, it is also intimately tied to the issues involved in successful conversation and one’s ability to elicit cooperative behavior from others. To formalize these concepts, H.P. Grice posited his Cooperative Principle and associated conversational maxims proposing that you should contribute to the conversation only what is required, at the appropriate stage in the conversation as accepted by the direction of the discussion in which you are engaged [52]. In considering the issues involved in construction of languages Valter Tauli also proposed a number of similar principles. Both sets of principles are outlined below (adapted from Traunmüller [125:2]).

Grice’s Conversational Maxims state:

1. Be as informative as required by the context, and no more.
2. State only what you believe to be true and for which you have adequate evidence.
3. Be relevant.
4. Avoid ambiguity and obscurity; be brief and orderly.

Tauli’s Principles of Language Planning state:

1. Convey everything intended.
2. Where semantic confusion may arise, ensure expressions are sufficiently distinct.
3. Convey only what is necessary.
4. Use redundancy where needed.
5. Keep the expression as short as possible.
6. The more frequent an expression, the shorter it should be.

Grice’s maxims are frequently referred to as Quantity, Quality, Relation, and Manner, shown as one through four above. To understand the relationship between the conversational maxims and language planning, consider that “if, in a conversation, you
want to fulfill the maxims of appropriate quantity, quality, and relation, you have to say no less and no more than what is required. In order for this to be fulfilled, the language must give you the freedom to choose a more or less specific expression according to the circumstances. The corresponding principle of language planning is known as the principle of facultative precision” [125:3]. An interesting comparison study of the grammars of many modern languages can be found in Joseph Greenberg’s paper entitled “Some Universals Of Grammar With Particular Reference to the Order of Meaningful Elements.” The results of this paper help to illustrate a number of common trends that occurred in the development of natural languages, and it also shows how the grammars of those languages may constrain the speaker.

While Grice’s conversational maxims seem logical, they are not always followed in the course of everyday conversation. One of the primary reasons for this departure is the issue of politeness. The work of Brown and Levinson [15] suggests that people will depart from the maximal use of speech to exploit language to meet their goals and also to handle the issue of face [49]. There are two dimensions to face. Positive face refers to an individual’s “desire to be appreciated as a social person” [111], while negative face “refers to the desire to see one’s actions unimpeded by others” [111]. Another perspective on these issues can be identified in Leech’s Politeness Principle and it’s associated maxims (adapted from Slemrouck [111]).

Negative Politeness consists of:

1. Tact Maxim – minimize cost to other,
2. Generosity Maxim – maximize benefit to self,
3. Approbation Maxim – minimize dispraise of other,
4. Modesty Maxim – minimize praise of self,
5. Agreement Maxim – minimize disagreement between self and other, and

Positive Politeness, on the other hand, consists of:
1. Tact Maxim – maximize benefit to other,
2. Generosity Maxim – maximize cost to self,
3. Approbation Maxim – maximize praise of other,
4. Modesty Maxim – maximize dispraise of self,
5. Agreement Maxim – maximize agreement between self and other, and

The notion of politeness is important because it shows that the communications employed in the context of a conversation are not solely for the purpose of information transfer. A similar idea motivated the early work of J.L. Austin. Austin attempted to clarify the distinction between a constative and a performative. A constative is an utterance that represents some state of affairs that can be judged as true or false (essentially a statement), whereas a performative is an utterance that produces some social effect by virtue of its being spoken. The notion of fit is frequently used to clarify the distinction. A constative is true if it accurately depicts the state of the world. If this is the case, then the constative is said to fit. Otherwise, it does not fit, and the constative is false. In the case of a performative, if the utterance alters the state of the world such that the new state conforms to the act, then the world fits to the performative. Notice that a constative fits the world, whereas the world fits to the performative.

There may be any number of reasons why the utterance of a certain performative may be unsuccessful (infelicitous, unhappy). The following list was proposed by Austin but was not suggested as a complete typology.
1. Misfires – performatives that are unhappy due to external circumstances
   a. misinvocation – conventions do not exist or were used inappropriately
      i. non-plays – conventions do not exists
      ii. misapplication – convention incorrectly applied
   b. misexecution – conventional procedures are not carried out completely
      i. flaws – conventions not performed by all parties
      ii. hitch – conventions incomplete

2. Abuse – performatives that are unhappy due to internal circumstances
   a. insincerities – performative uttered without commensurate intentions
   b. non-fulfillment – performative uttered but intentions not fulfilled

It is interesting to note that internal circumstances essentially represent the intentions of the speaker, his mental state at the time the performative is uttered. So, an utterance of the form “I promise to give you this book,” will be an abuse, if my intentions were not to do so. This notion ties the study of performatives not only to the structure of the language, but also to the psychology (intentions) of the speaker. The external circumstances, which combined with the internal circumstances complete what is called the total speech situation, deal with conventionally accepted procedures. For instance, though a man may state the proper words to consummate a marriage, if that man does not have appropriate authority, a misapplication has occurred.

Ultimately, Austin abandons the distinction between constatives and performatives, and opts for a theory in which the concepts are two dimensions of the same speech act. This Speech Act Theory considers speech as having meaning, conventional force, and non-conventional force. The meaning is what was previously associated with the notion of a constative, while the conventional and non-conventional forces are aspects of the
performative dimension. A more detailed decomposition of the Austinian speech act appears as follows:

1. Locutionary act – the meaningful utterance,
   a. Phonetic act – the noises of the utterance (phonemes),
   b. Phatic act – the order of the phonemes – constitutes creation of a pheme when the utterer understands proper use of the language, and
   c. Rhetic act – use of the pheme in some context to perform some action,
2. Illocutionary act – the conventional (performative) force of the meaningful utterance, and
3. Perlocutionary act – the non-conventional effect brought about by the utterance.

To better understand the aspects of Austin’s speech act, it is useful to compare his structure to the later formulation postulated by John R. Searle. Searle envisioned a distinction between the intention to represent versus the intention to communicate. To this end, he replaced Austin’s locutionary act with something Searle called a propositional act. The decomposition of the Searlian speech act is:

1. Utterance act – a speech act devoid of meaning,
2. Propositional act
   a. Reference act – complete speech act refers to something and,
   b. Act of predication – not a separate speech act, has no reference,
3. Illocutionary act – the conventional (performative) force of the meaningful utterance and,
4. Perlocutionary act - the non-conventional effect brought about by the utterance.

It is important to understand that the propositional act only has meaning when it refers to something. Furthermore, that meaning is a result of the intention of speaker, the context, and any accepted social conventions. This notion is consistent with the work of Brentano who suggested that all mental acts are directed towards some object. Further
generalization of this concept can also be seen in the work of Husserl in describing the range of objectifying acts. As described by Smith, such acts include:

1. Acts directed towards individual things, events, processes, etc., and towards the parts and moments of these;
2. Acts directed towards species or essences, and towards ideal objects such as numbers; and
3. Acts, above all acts of judgement, directed towards Sachverhalte or states of affairs. [115:3]

Returning to the notion of representation versus communication, it is useful to appeal to example. Searle envisions an American soldier captured by Italians. By speaking a line of prose from a German song, the American desires to fool the Italians into thinking he is German. This is the only German sentence the soldier knows, and it literally speaks of blossoming lemon trees, not his status as a German. Thus, the soldier’s intention to represent (the location of blossoming lemon trees) did not match his intention to communicate (his status of being German), though the illocutionary act, and hopefully the perlocutionary act (of his release) may still succeed. One may take this idea a step further and suggest that it is only the intent to communicate that matters, not the actual format of the representation. This may arguably be the basis for metaphor, irony, and other parasitic speech acts (see Halion [58]).

2.1.6 Agent Communication Languages

Certain additional aspects of Searle’s work on speech act theory have recently found their way into the realm of computer science, specifically in the design of agent communication languages. In the previous section, we discussed Searle’s decomposition of a speech act into a number of specific dimensions. In this section, we briefly examine a typology of speech acts, and also explore the concept of felicity conditions. Following
the introduction of these two topics, we look at an agent architecture based on rational behavior, and how this architecture led to development of particular agent communication languages.

One possible classification of speech acts, due to Searle, suggests the following categories – the category name is followed by a prototypical example and then a definition:

1. Representative – assertion – used to communicate the truth value of some proposition.
2. Directive – request, question – used to attempt to motivate some action from the receiver.
3. Commisive – promise, threat – used to commit the speaker to a future course of action.
4. Expressive – apology, welcome – used by a speaker to describe his mental state.
5. Declarative – marrying, christening – used by speaker, along with some accepted social convention, to affect a change in some institutional state of affairs.

The list above is only one particular view, and many other classifications (and classes) can be found in the literature. The important point is that such classifications exist and are convenient for describing properties of generic types of speech acts. One such property of speech acts is that of felicity conditions. Felicity conditions are states that must be obtained for the successful completion of a speech act. These conditions have been classified into preparatory, prepositional, sincerity, and essential types. For example, the conditions for questioning appear as follows [106]:

\[ S = \text{speaker}, \ H = \text{hearer}, \ P = \text{the proposition expressed in the speech act} \]

1. Preparatory 1: S does not know the answer, i.e. for a yes/no question, does not know whether P is true or false; for an elicitative or WH-question, does not know the missing information.
2. Preparatory 2: It is not obvious to both S and H that H will provide the information at that time without being asked.

3. Propositional: Any proposition or propositional function.

4. Sincerity: S wants the information.

5. Essential: The act counts as an attempt to elicit this information from H.

With these notions in mind, we are now in a position to consider the BDI agent architecture. The acronym BDI stands for belief, desire, and intention and conveys the emotional foundation on which these agents are based. Not all authors agree on the necessity of these three mental states. However, even in the papers where alternative mental states are introduced, the authors usually feel compelled to provide a mapping back to BDI. As a quick way of highlighting some additional possibilities and explaining their meaning, we quote the list of mental concepts identified by Huhns [62:14].

1. Beliefs – Characterize what an agent imagines the state of the world to be, that is, how the agent represents the state of the world.

2. Know-how – Characterizes what the agent can really control in its environment.


4. Desires – Describe the agent’s preferences and may sometimes have a motivational aspect.

5. Intentions – Characterize the goals or desires that the agent has selected to work on.

To help classify BDI agents in the context of other agent models and their environment, Huhns and Singh [62:12] also present the following classes:

1. Behaviorism or positivism – considers only the direct behaviors of agents and their environments – agents have no mental states (no BDI) – multi-agent systems have no social states such as commitments – causality not first class – the universe involves plain sequences of events.

2. Subjectivism – there are intentions and commitments, but they are as represented in an agent – the universe can have causation but only as represented in an agent.
3. Realism – the universe has causes – the agents have intentions and beliefs – multi-agent systems exist and involve the agent’s commitments.

In dealing with the notions of knowledge and belief, it is necessary to question what one really knows or can believe based on his knowledge. Three interesting questions arise:

1. Does an agent “know” all that follows from its beliefs and logic? That is, does it possess tautological closure.
2. Does an agent “know” that it knows a given fact? That is, does it demonstrate positive introspection?
3. In a similar fashion, does an agent “know” that it does not know something? Does it demonstrate negative introspection?

These are not the only issues that can be addressed, but they do suggest some of the problems encountered in the study of mental states. Accordingly, Shoham [107:335] discusses some desirable properties of agents with respect to their mental states. These properties include:

1. Internal consistency – both beliefs and obligations are internally consistent.
2. Good faith – agents commit only to what they believe themselves capable of, and only if they really mean it.
3. Introspection – agents are aware of their obligations.
4. Persistence of mental state – agents have perfect memory of, and perfect faith in their beliefs, and only let go of a belief if they learn a contradictory fact.

These desirable properties correspond closely to the Gricean Maxims and show up shortly in our discussion of particular agent communication languages. Without delving into the detailed formalism behind BDI agent models, it is still useful to examine the high-level components of such systems. As outlined by Suchman, et al. [120:416], an agent supporting the concept of mental states should maintain notions such as:

1. A social reasoning mechanism that enables an agent to reason about others, using information about their goals, actions, resources, and plans.
2. An external description which is a per agent data structure where the agent stores the information it has about others.

3. The goals an agent wants to achieve.

4. The actions the agent is able to perform.

5. The resources an agent has control over.

6. The plans an agent has, using any actions and resources, in order to achieve a certain goal – the actions and resources do not necessarily belong to his own set of actions and resources, and therefore an agent may depend on others in order to carry on a certain plan.

2.1.6.1 Description of the FIPA-ACL

The Foundation for Intelligent Physical Agents (FIPA) expresses its ultimate goal as promoting “the success of emerging agent-based applications, services and equipment” [38]. To achieve this goal, FIPA has defined seven normative references including Agent Management, Agent Communication Language (ACL), Agent-Software Integration, Human-Agent Interaction, Agent Security Management, Agent Management Support for Mobility, and Ontology Service. In addition, FIPA provides four informative documents specifying industrial applications of agent technologies. Finally, a developers guide is provided explaining proper usage of the material contained in the normative and informative references. In this section, we focus on the ACL document that specifies a “set of message types and the description of their pragmatics” [37].

A FIPA-ACL message takes the form of an s-expression and appears as shown below.

```
(communicative_act
  :sender       <name of the sender>
  :receiver     <names of the intended recipients>
  :content      <the content encoded in the language specified by
                 :language>
  :reply-with   <a unique conversation identifier>
  :in_reply_to  <a unique conversation identifier>
  :envelope     <message transport information – not part of the ACL
                 specification>
```

Note that the angle brackets contain a description of the data placed in the corresponding fields. The angle brackets are not part of the syntax of a FIPA-ACL message.

Of all the parameter types listed, the only FIPA mandated argument is :receiver. At a higher level, FIPA mandates five requirements for compliance to the ACL specification [37]:

1. When an agent receives a message it does not understand, it should reply with the “not-understand” communicative act. Likewise, all agents should be prepared to receive such a reply in response to any message they send.

2. FIPA places no restrictions on the number or types of messages implemented by an agent, only that those messages implemented must conform to the specification.

3. More specifically, communicative acts of the same name as those defined in the specification must conform to the provided interpretation.

4. And, communicative acts of different names should not have the same meaning as existing acts.

5. Finally, an agent must be able to create a syntactically well-formed message.

The content language is whatever language the agent chooses to express the content of its messages. FIPA places no restrictions on the type of language used except that the language must support the representation of propositions, objects, and actions. A proposition is a description of the veracity of a sentence; an object represents an identifiable entity; and an action is an activity that the agent can perform. Decision on which language to use will likely be based on the languages understood by the other
agents in the conversation. While this choice can be negotiated, there is still a necessity to standardize a language for new conversations. For this purpose, FIPA insists that all agents support the s-expression notation based agent management content language. The requirement is made firm by the following statement:

“A compliant agent is required to exercise the standard agent management capabilities through the use of messages using the agent management content language and ontology. The language and ontology are each denoted by the reserved term $\text{fipa-agent-management}$ in their respective parameters.” \[37\]

FIPA defines two, pragmatically equivalent, types of communicative acts: primitive and composite. A primitive communicative act is atomic, composed only of itself. A composite communicative act is composed of one or more underlying primitive or composite acts. FIPA identifies three types of composite communicative acts:

1. The first communicative act references the second. The act, “I request you to inform me if that item is for sale” is an example of a $\text{query-if}$.\[398\]

2. A set of communicative acts to be performed in sequence. Denoted by $a;b;c$ where $a$ then $b$ then $c$ are to be performed in order.

3. A set of communicative acts only one of which should be performed. The choice, denoted by $a|b|c$, where only one can be performed.

FIPA has chosen to classify its messages into five categories: information passing, requesting information, negotiation, action performing, and error handling. Before looking at an example of a particular message however, it is necessary to introduce the Semantic Language (SL). Using first order modal logic with identity, SL attempts to model the beliefs, uncertainties, and choices (goals) of agents. These three primitive mental attitudes are represented as logical propositions using the following formalism:

1. $\text{B}_i p$ represents the fact that some agent, $i$, implicitly believes the proposition, $p$.

2. $\text{U}_i p$ represents the fact that the agent, $i$, is not certain about the truth of the proposition $p$, but thinks it is likely to be true.
3. \( C_i p \) represents the fact that the agent, \( i \), desires that the proposition, \( p \), currently holds.

FIPA also defines the four operators \textit{Feasible}, \textit{Done}, \textit{Agent}, and \textit{Single} to operate on complex plans or actions. The fact that some action, \( a \), can take place, and that upon completion some proposition, \( p \), will be true, is represented by the expression \textit{Feasible}(\( a \), \( p \)). The expression \textit{Done}(\( a \), \( p \)) means that the action, \( a \), has occurred, and that proposition, \( p \), was true just prior to the occurrence. As a shorthand notation for \textit{Done}(\( a \), \( true \)), FIPA permits the simpler \textit{Done}(\( a \)). Given some action expression, \( a \), if we wish to express that only a single agent is performing or will perform those actions, we write \textit{Agent}(\( i \), \( a \)). Finally, \textit{Single}(\( a \)) is used to determine whether a given action expression represents a sequence. That is, \textit{Single}(\( a \)) is true if the action is either a single event, or is a nondeterministic choice (e.g., \( a|b|c \)) in which all component actions are single events (e.g., \( a \), \( b \), and \( c \) are all single events).

When an agent has committed itself to the achievement of some goal proposition, and will continue to progress towards this goal until either succeeding or realizing it’s impossibility, that goal is said to be a persistent goal of the agent. The notion of goal persistence is represented by the formula \( PG_i p \) which states that agent, \( i \), has proposition, \( p \), as a persistent goal. Along these same lines, \( I_i p \) signifies that agent \( i \) has intention to act upon the persistent goal \( p \).

The Gricean maxims, discussed in section 2.1.4, and feasibility conditions, discussed above, arise when defining a particular communicative act. Before an agent can perform a communicative act, it must ensure the validity of the feasibility preconditions (FP). Once the preconditions are satisfied, the agent is free to include the act in whatever planning mechanism it employs. There are two types of \textit{feasibility preconditions}. 

Ability preconditions specify certain characteristics or beliefs that the agent must possess to perform a given communicative act. Context-relevant preconditions corresponding to the Gricean maxims of quantity and relation, specify that an agent should perform only those acts that are relevant in the current context, and nothing more. The decision to choose a particular act is governed by what is referred to as the *rational effect* (RE). Though the rational effect desired by the performing agent may not come to pass, it is the expectation of its occurrence that drives the agent to choose that particular action.

To make the concepts introduced above more concrete, we now examine the FIPA-defined *confirm* message. The confirm message is an information passing message that is passed from a sender to a receiver when the sender wishes to inform the receiver that a given proposition is true. More specifically, the sender believes the proposition is true, believes that the receiver is uncertain of the truth value, and for whatever reason desires that the receiver comes to believe that the proposition is true. Whether the receiver actually chooses to believe the proposition, is not dictated by the standard; it is the choice of the receiver.

The confirm message is defined as follows:

\[
<i, \text{confirm}(j, \varphi)>
\]

\[
\text{FP: } Bi\varphi \land BiU_j\varphi
\]

\[
\text{RE: } Bj\varphi
\]

This formalism states that the communicative act *confirm* was sent from agent \(i\) to agent \(j\) containing the prepositional content, \(\varphi\). Furthermore, the feasibility precondition of the act is that agent \(i\) believes \(\varphi\) to be true and that agent \(i\) believes agent \(j\) is uncertain of the truth of the proposition. Finally, the rational effect desired by agent \(i\) is that agent \(j\) comes to believe proposition \(\varphi\).
A more complicated example is seen in the definition of the inform-ref act. We begin by showing the SL representation:

\(<i, \text{inform-ref}(j, 1x \delta(x))> \equiv <i, \text{inform}(j, 1x \delta(x) = r1)> | \ldots | <i, \text{inform}(j, 1x \delta(x) = rk)>\)

FP: Brefi 1x δ(x) ∧ ∼Bi(Urefj 1x δ(x) ∨ Brefj 1x δ(x))
RE: Brefj 1x δ(x)

Starting at the top, we see that an agent who plans an inform-ref act will actually perform one of \(k\) (where \(k\) may be infinity) possible inform acts. Of course, the agent does not actually plan each of the possibly infinite number of acts. Instead he plans to send the information, the value of \(x\), that makes \(δ(x)\) true. And, that is exactly what the notation, \(1x δ(x)\), states. The Greek iota symbol is read “the \(x\) (which is) \(δ\).” Or better yet, the value of \(x\) that makes the proposition \(δ(x)\) true. So, the feasibility precondition states that agent \(i\) believes it knows the correct value of \(x\) and agent \(i\) does not believe that agent \(j\) (the receiving agent) either knows the value of \(x\) or is even uncertain of the value of \(x\). The rational effect should be that the receiving agent comes to believe that it too knows the value of \(x\). To illustrate a potential use of this communicative act we present an example.

Agent \(i\) requests \(j\) to state who is the author of “Oliver Twist”:

(request
  :sender i
  :receiver j
  :content
    (inform-ref
      :sender j
      :receiver i
      :content (iota ?x (author “Oliver Twist” ?x))
      :ontology bible
      :language sl
    )
  )
  :reply-with query0
  :language sl
)
Agent \( j \) replies:

\[
\text{(inform)} \\
\text{sender} \; j \\
\text{receiver} \; i \\
\text{content} \; (= \; (\text{author} \; \text{"Oliver Twist"} \; ?x) \; \text{"Charles Dickens"}) \\
\text{ontology} \; \text{bible} \\
\text{in-reply-to} \; \text{query0} \\
\]

This example illustrates a number of important issues. First, it provides a feeling for the actual usage of the FIPA communicative acts in a conversation between two agents. Next, it demonstrates the conversation tags :reply-with and :in-reply-to. Finally, it shows the population of the content field with the previously defined SL language. The reader should be careful to note that agent \( j \) planned the \textit{inform-ref} act.

In addition to the individual speech acts, two of which we have outlined above, FIPA also defines a protocol or “typical pattern of message exchange” [37]. Protocols serve to relieve agent implementations of the burden of complex emotional modeling. While their use is not mandated, a conversational protocol can greatly simplify the internal processing required to engage in meaningful multi-agent discourse. The most notable examples may be the predefined commerce (bidding) protocols. If the tasks involved in these negotiations are predefined, then the agent does not have to “choose” the most appropriate communicative act – it is chosen for him. This is not to say that all of the work can be accomplished in a lookup fashion; the agent is still responsible for choosing the appropriate values and propositional content. But, it is relieved from the potentially onerous task of determining the expected response type for each message it receives.
As an example, Figure 2-2 illustrates the FIPA-request protocol. The protocol diagram is adapted from a representation from FIPA [37]. However, we have chosen to represent the actions using a message passing notation rather than the hierarchical diagrams used by FIPA. Viewing the illustration, we see that the request protocol begins with Agent A sending a *request* action to Agent B. Upon receipt of this message, Agent B may perform any of three mutually exclusive activities. First, if the message was, for some reason, *not understood*, then Agent B should send a *not-understood* message back to Agent A. As noted by FIPA [37], Agent A must not reply in turn with another not-understood, otherwise an infinite loop occurs. Agent B’s second alternative is to *refuse* the request and provide some explanation for doing so. Finally, Agent B may *agree* to perform the action at some, possibly specified, later time. In the illustration, the choice among these three courses of action is shown by using a right brace to wrap the three alternative message lines. If Agent B chooses the last option, then it commits itself to performing the requested action and moves into the state represented by circle 2. As shown in the diagram, agreeing to perform the action causes a state transition from state 1...
to state 2, as well as the propagation of the agree message. The break in the bold, vertical timeline shows that the eventual informs occur at some time after the original agree. There are, in fact, three possible outcomes after agent B has accepted the task. First, the agent may fail to perform the task, in which case it must notify agent A with a failure message. Second, agent B may simply notify agent A that it has completed (is Done) with the requested action. Finally, agent B may notify agent A that the action has completed with some specified result.

Another protocol provided by the FIPA specification is the contract net protocol. Figure 2-3 shows the state machines of both agents and the messages passed between them over time. Note that in accordance with the goals outlined by FIPA, we are not insisting that the internal state machine of the agents follow the path outlined in the figure. However, the inclusion of state machine style information serves well to clarify the progression of the protocol over time. The contract-net protocol begins with the manager agent transmitting a call-for-proposals to any number of contractor agents. We have illustrated the interactions only between the manager and a single contractor; the extension to multiple contractors is immediate.
Though not specified in the protocol description, it is instructive at this point to mention the :reply-with, :in-reply-to, and :conversation message parameters. These parameters are used to enable multi-agent and context abbreviated message content. Using the :conversation parameter, an agent can maintain state across a number of communications by simply identifying each message as belonging to the identified conversation (a unique identifier). The :reply-with and :in-reply-to parameters provide further assistance by permitting shortened answers. If I ask the question, “what is the current time?” a reasonable answer would be “10:00pm UTC.” However, if we are maintaining multiple conversations (possibly with the same agent) then it is not clear (without additional processing) that the answer, “10:00pm UTC,” is a response to any particular question. In this case, only the full answer, “the current time is 10:00pm UTC” would be an unmistakable answer to our question. Further information regarding the proper utilization of these parameters can be found in annex A of the FIPA specification [37].
Continuing our discussion of the contract-net protocol, we see that the agent receiving the \textit{cfp} message has three possible courses of action: replying that is does not understand, refusing with an explanation, or providing a proposal to perform the requested task. As shown, to avoid having the manager wait indefinitely to receive all proposals, any proposals returned past the given deadline are considered late and will not be accepted. Once the manager has received all of the proposals, it accepts whichever ones meet its criteria and sends acceptance messages. This protocol efficiently simplifies the processes involved in negotiation and can be extended in any number of ways. The FIPA-iterated-contract-net protocol is one such extension. Without providing a corresponding illustration, we note that the iterated version of the protocol adds the possibility for the manager to collect the proposals, revise its original problem statement, and reissue an updated \textit{cfp}. There is no limit to the number of revisions, but the process is set to terminate when all contractors refuse, the manager accepts a set of proposals, or the manager cannot reach a satisfactory agreement and refuses all bids. Further refinements of the call-for-proposal and bidding process are possible. FIPA illustrates two auction types, the English and the Dutch versions. We only mention here that the English auction is the familiar protocol under which the auctioneer continues to raise the price of the article until no more bids are received, while the Dutch auction starts high and iteratively lowers the price. An additional and widely acclaimed bidding technique discussed by Rosenschein and Zlotkin [105] is Vickery’s Mechanism. The mechanism proceeds with all contractors making their bids, and the lowest bid wins the contract. However, the winner gets the price submitted by the second lowest bidder. By separating who wins the bidding from how much they bid, the technique strives to negate any advantages gained
by overbidding or underbidding. Where honesty cannot be guaranteed, this technique may prove far superior.

2.1.6.2 KQML

The Knowledge Query and Manipulation Language (KQML) is an agent communication language developed by the DARPA Knowledge Sharing Effort (KSE). KQML serves much the same purpose as the FIPA ACL but predates the FIPA effort and differs in a number of particulars. Our coverage of KQML does not detail each feature as explicitly as was done for FIPA ACL; however, we attempt to highlight the distinctions between the two languages. The DARPA KSE suggests four levels of interface agreement for successful interoperation of agent-based systems [33]:

1. Transport: how agents send and receive messages,
2. Language: what individual messages mean,
3. Policy: how agents structure conversations, and
4. Architecture: how to connect systems in accordance with constituent protocols.

With these levels defined, DARPA-KSE places the KQML specification within the language domain. The primary goal of the language is outlined by defining KQMLs intended use as “a high-level language to be used by knowledge-based system[s] to share knowledge at run time” [33]. Knowledge-based systems are typically autonomous agents whose goals need not converge or even agree. Each agent is said to manage a virtual knowledge base (VKB) containing that agents’ beliefs and goals. While the DARPA-KSE places no internal restrictions on the implementation of the agent, it does insist that the outward appearance of an agent cause the illusion (true or not) that there is a real knowledge base hidden beneath. The beliefs contained in that VKB maintain information regarding the outside environment, the VKBs of other agents, and a representation of the
agent’s own knowledge. The desires of the agent, the affects it wishes to have on other agents or its outside environment, are encoded as the goals within the VKB.

KQML messages borrow their syntax from the Common Lisp Polish prefix notation. Messages, or performatives as they are more formally called, are Lisp structured ASCII strings containing the performative name as well as any number of parameter name/value pairs. The following example illustrates the format of a KQML message and also clearly demonstrates the KQML influence on the FIPA ACL standard:

```lisp
(tell
  :sender A
  :receiver B
  :in-reply-to mid0
  :reply-with mid1
  :language Prolog
  :ontology plants
  :content "tree(orange, angiosperm)"
)
```

The similarities with FIPA ACL are significant. However, while FIPA ACL and KQML, both allow the addition of new communicative acts, the `tell` act is a reserved KQML act but is not present in the standard FIPA ACL specification. In KQML terminology, the `tell` performative is roughly equivalent to the FIPA `<i, inform(j, Bip)>` act. The intention of the message is to tell the recipient that the given “:content” is contained in the sender’s VKB.

In a fashion similar to FIPA communicative acts, the performatives of KQML are also subdivided into categories based on their usage. The performative categories include discourse, intervention and mechanics, and facilitation and networking. Table 2-2, adapted from Labrou and Finin [71], places each performative into its appropriate category, and provides a brief definition.
### Table 2-2. KQML performatives

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discourse</td>
<td>ask-if</td>
<td>S wants to know if the :content is in R’s VKB</td>
</tr>
<tr>
<td></td>
<td>ask-all</td>
<td>S wants all of R’s instantiations of the :content that are true of R</td>
</tr>
<tr>
<td></td>
<td>ask-one</td>
<td>S wants one of R’s instantiations of the :content that is true of R</td>
</tr>
<tr>
<td></td>
<td>stream-all</td>
<td>multiple-response version of ask-all</td>
</tr>
<tr>
<td></td>
<td>Eos</td>
<td>the end-of-stream marker to a multiple response (stream-all)</td>
</tr>
<tr>
<td></td>
<td>Tell</td>
<td>the sentence is in S’s VKB</td>
</tr>
<tr>
<td></td>
<td>Untell</td>
<td>the sentence is not in S’s VKB</td>
</tr>
<tr>
<td></td>
<td>Deny</td>
<td>the negation of the sentence is in S’s VKB</td>
</tr>
<tr>
<td></td>
<td>Insert</td>
<td>S asks R to add the :content to its VKB</td>
</tr>
<tr>
<td></td>
<td>Uninsert</td>
<td>S wants R to reverse the act of a previous insert</td>
</tr>
<tr>
<td></td>
<td>delete-one</td>
<td>S wants R to remove one matching sentence from its VKB</td>
</tr>
<tr>
<td></td>
<td>delete-all</td>
<td>S wants R to remove all matching sentences from its VKB</td>
</tr>
<tr>
<td></td>
<td>Undelete</td>
<td>S wants R to reverse the act of a previous delete</td>
</tr>
<tr>
<td></td>
<td>Achieve</td>
<td>S wants R to make something true of its physical environment</td>
</tr>
<tr>
<td></td>
<td>Unachieved</td>
<td>S wants R to reverse the act of a previous achieve</td>
</tr>
<tr>
<td></td>
<td>Advertise</td>
<td>S wants R to know that S can and will process a message like the one in :content</td>
</tr>
<tr>
<td></td>
<td>Unadvertised</td>
<td>S wants R to know that S cancels a previous advertise and will not process any more messages like the one in the :content</td>
</tr>
<tr>
<td>Intervention and Mechanics</td>
<td>Subscribe</td>
<td>S wants updates to R’s response to a performative</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>S considers R’s earlier message to be malformed</td>
</tr>
<tr>
<td></td>
<td>Sorry</td>
<td>S understands R’s message but cannot provide a more informative response</td>
</tr>
<tr>
<td></td>
<td>Standby</td>
<td>S wants R to announce its readiness to provide a response to the message in :content</td>
</tr>
<tr>
<td></td>
<td>Ready</td>
<td>S is ready to respond to a message previously received by R</td>
</tr>
<tr>
<td></td>
<td>Next</td>
<td>S wants R’s next response to a message previously sent by S</td>
</tr>
<tr>
<td></td>
<td>Rest</td>
<td>S wants R’s remaining responses to a message previously sent by S</td>
</tr>
<tr>
<td></td>
<td>Discard</td>
<td>S does not want R’s remaining responses to a previous (multi-response) message</td>
</tr>
<tr>
<td>Category</td>
<td>Name</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Facilitation and Networking</td>
<td>register</td>
<td>S announces to R its presence and symbolic name</td>
</tr>
<tr>
<td></td>
<td>unregister</td>
<td>S wants R to reverse the act of a previous register</td>
</tr>
<tr>
<td></td>
<td>forward</td>
<td>S wants R to forward the message to the :to agent (R might be that agent)</td>
</tr>
<tr>
<td></td>
<td>broadcast</td>
<td>S wants R to send a message to all agents that R knows of</td>
</tr>
<tr>
<td></td>
<td>transport-address</td>
<td>S associates its symbolic name with a new transport address</td>
</tr>
<tr>
<td></td>
<td>broker-one</td>
<td>S wants R to find one response to a &lt;performative&gt; (some agent other than R is going to provide that response)</td>
</tr>
<tr>
<td></td>
<td>broker-all</td>
<td>S wants R to find all responses to a &lt;performative&gt; (some agent other than R is going to provide that response)</td>
</tr>
<tr>
<td></td>
<td>recommend-one</td>
<td>S wants to learn of an agent who may respond to a &lt;performative&gt;</td>
</tr>
<tr>
<td></td>
<td>recommend-all</td>
<td>S wants to learn of all agents who may respond to a &lt;performative&gt;</td>
</tr>
<tr>
<td></td>
<td>recruit-one</td>
<td>S wants to get one suitable agent to respond to a &lt;performative&gt;</td>
</tr>
<tr>
<td></td>
<td>recruit-all</td>
<td>S wants to get all suitable agents to respond to a &lt;performative&gt;</td>
</tr>
</tbody>
</table>

Starting with the discourse performatives, we see a number of ask-* variations. The ask-if performative is straightforward and simply requests information concerning the existence of the given :content in the receiver’s VKB. The more interesting discourse performatives are ask-one, ask-all, stream-all, tell and eos, along with the intervention and mechanics performatives next, rest, standby, ready and discard. Consider a question with multiple answers; for example, “Tell me all positive number less than 5.” If such a question is asked, the reply could take on many different forms. First, the answering agent could simply provide an implementation specific structure (such as a list) containing all of the answers. Alternatively, the individual answers could be returned one at a time until the set has been exhausted. Furthermore, the intention of the questioning agent may be to only obtain a single representative element rather than the complete set.
The previously mentioned performatives provide for each of these alternatives, as well as various permutations on timing.

The *ask-one* message requests a single element from the set of correct answers. Similarly, *ask-all* requests the complete set. The performative *stream-all* also requests the complete answer set, but unlike the *ask-all* performative that sends the answer in a single message, *stream-all* sends one message (*tell*) per element of the answer set. To distinguish the end of the stream, the *eos* message is sent to complete the communication. Further refinements to the process can be achieved through the use of the intervention and mechanics performatives. If the questioning (asking) agent sends a *standby* message along with its question, then the answering agent only returns a *ready* notification but retains the actual answer(s) until the original agent requests it. When the request for the answer set arrives, it can be for one element of the answer set at a time through the use of *next* or for the as yet unsent elements through the use of *rest*. The *stream-all*, *eos*, *tell*, *standby*, and *next* performatives together provide a functionality similar to that of enumerators in an object-oriented language, whereas *ask-all* and *tell* provide the procedural API-like return of an array of values through a parameter. Finally, *discard* is used to tell the answering agent that the questioning agent is not interested in any remaining elements of the result set.

Another interesting set of performatives includes *insert*, *delete-one*, *delete-all*, and *uninsert*. These messages allow the sending agent to modify (add or remove) facts from the VKB of the recipient agent. That is, *insert* is used by an agent when it wishes to add a new fact to another agent’s VKB, whereas *delete-* is used to remove one or all facts (meeting some criteria) from another agent’s VKB. Obviously, utilization of these
messages under the assumption that they are always successful contradicts the notion of autonomy with possibly divergent goals. To alleviate this situation, agents are only subject to the results of these messages if they so elect, by advertising that intention. Here, advertising is taken quite literally to mean the placement of an advertisement with a facilitator.

As mentioned in the last paragraph, there exists the notion of a facilitating agent. “In each domain of KQML-speaking agents there is at least one agent with a special status called facilitator that can always handle the networking and facilitation performatives. Agents advertise to their facilitator, i.e., they send advertise messages to their facilitators, thus announcing the messages that they are committed to accepting and properly processing” [71]. Furthermore, “agents can use their facilitator either

- to have their queries properly dispatched to other agents, using recruit-one, recruit-all, broker-one or broker-all, or
- to send a recommend-one or a recommend-all to get the relevant advertise messages and directly contact agent(s) that may process their queries”

It is worthwhile to note that the KSE KQML networking and facilitation performatives and the notion of a facilitator, together provide much the same functionality as defined in the FIPA agent management specification. Details on the FIPA approach can be found in the FIPA 98 Specification [38].

2.2 Programming Language Design

Object-oriented techniques [79][77][34][44] have proven to be a step in the right direction for large-scale software development. However, Ossher and Harrison suggest that "if object-oriented technology is to be successfully scaled from the development of independent applications to development of integrated suites of applications, it must relax its emphasis on the object” [95:1]. This sentiment seems to be shared to some degree by
many current researchers. A quick review of object-oriented concepts and an introduction to some new trends in language design will help to clarify the underlying rationale.

Fisher and Mitchell [34:2] state that an object-oriented language should provide at least the following features:

1. Dynamic Lookup – the run time type of an object, rather than its static type, determines the proper method to be executed – also known as polymorphism,

2. Subtyping – given some object A that exposes at least as much functionality as another object, B, we may use A in any context expecting B – also known as subtype substitutability,

3. Inheritance – ability to define complex objects in terms of simpler ones, and

4. Encapsulation – limiting access to the internal state of an object.

Note that the third characteristic mentioned above is not a mandatory requirement for all object-oriented languages. The language, Self [129], uses delegation rather than inheritance as its style of differential definition. In fact, inheritance is one of the most highly contested aspects of object-oriented programming, and many authors feel its use is frequently unwarranted [65][44].

The implementation of dynamic lookup also provides a differentiator in the classification of object-oriented languages. As presented by Harrison [59], the three major models for object-oriented languages are the common model, the generic function model, and the brokered message model. Briefly, in the common model, a name binding system (linker or loader) is used to publish the interface of a class to describe the functions supported by instances of that class. In this way, message delivery (method invocation) needs simply go to the object identified to determine its implementation. Languages such as C++ and Java use the common model. In the generic function model,
used by CLOS and Cecil, a registry of available methods and calling contexts is
maintained by the name-binding system. The delivery infrastructure identifies the correct
method in the repository based on the types of the arguments to the method call. This
distinguishes the common model from the generic model in that the latter supports the
notion of multiple-dispatch [34]. Finally, in the brokered message model, a method call
is explicitly formed as a message. That is, the delivery infrastructure acts to pass a
message between the consumer and the service supplier. The delivery can be performed
independent of either process both in time and space. As described by Harrison [59:4],
brokering facilitates "communication among applications written in different languages
and running in different environments in a network and makes the generic function model
widely accessible."

Subtyping is also a topic of much discussion. Madsen, et al. [79] attempt to make a
clear distinction between the notion of subtyping as opposed to subclassing. This
distinction is not made explicit in languages such as C++ that use classes as types. As
highlighted by Johnson and Foote [65], such languages "require that an object have the
right superclass to receive a message, not just that it have the right protocol. However, it
is also noted that languages with multiple inheritance can solve this problem by
associating a superclass with every protocol." An enlightening perspective is provided by
Nicolaou [93] where the author feels it is unfortunate that the class mechanism does not
prevent subclasses from introducing new methods inside class bodies, since this "would
have truly separated subtyping from implementation inheritance." Before continuing, it
is useful to review the purpose and characteristics of type systems.
Palsberg and Schwartzbach [97:4] state "the purpose of a type system is to allow the programmer to annotate programs with information about which methods are available in a given object, and to allow the compiler to guarantee that the error message NotUnderstood will never occur." The riddle of type systems, referred to by Thorup [124], provides an illuminating view into the problems presented to the language designer. The riddle states that the type system of any object-oriented language always reflects some tradeoff between covariance typing, full static typing, and subtype substitutability. As formalized in Madsen et al. [79] [80] it is not possible to maintain all three characteristics since support of any two mutually excludes the third. For example, support of covariance typing would imply that if method Fnc of class general had signature general::Fnc(general arg), then a subclass of general called subgeneral would expose the same method with signature subgeneral::Fnc(subgeneral arg). This does not occur in languages such as C++ and Java that do not support covariance typing; however, a methodology for enhancing Java is this fashion is presented by Thorup [124] through the use of virtual types. Subtype substitutability, as discussed above, would permit code of the form general x = new subgeneral(). Here we see that the base type is being used to hold an instance of the derived type. Now, since subgeneral::Fnc(subgeneral arg) only needs a qual(arg) = general for subtype substitutability, but needs qual(arg) = subgeneral for covariance typing, it is impossible to enforce static typing.

For the reasons presented above, language designers have made concessions in a number of different areas. Type systems of current languages can be classified by many distinct characteristics [85][79][34]. Strength of a type system indicates the amount of information conveyed by the type of an expression. In a weakly typed system, the type of
an expression provides little information. Smalltalk is an example of a weakly typed language. In a perfectly strong type system, the type of an expression would carry all required information and would permit all type checking to be performed at compile-time. In languages like Java, a hybrid approach is taken where the compiler does all that it can, and any remaining issues (such as casts) are handled by runtime checks. Closely related to the concept of strength is the safety of a type system. Safety is concerned with the temporal availability of the type information. In a safe language, a large amount of information is available at compile-time, and therefore a number of checks can be made to prevent coding errors. In a language such as Python, all type information is extrapolated at runtime, and the language is therefore unable to provide a high degree of safety. The notion of static versus dynamic checking provides us with another classifier. In a statically checked type system, type checking is performed at compile time, whereas in a dynamic system it is postponed until the line of code is executed (runtime). Finally, the distinction between early and late binding specifies when the type information is used to bind an expression to its appropriate implementation. In dynamic lookup (i.e., dynamic binding, dynamic dispatch, run-time dispatch) when a message is sent to an object at run time, the corresponding method is retrieved from that object’s method table [34]. This differs from static lookup where the type information available at compile time is actually used for locating the appropriate methods and improving runtime efficiency (C++). As a concrete example, Java is strongly-typed and supports late-binding. As such, it performs a number of type checks at compile time but determines the location of its code at runtime. This is what permits the use of the dynamic class loader functionality.
Having completed a quick review of object-oriented language issues, we are now in a position to examine a number of research proposals that seek to address specific deficiencies in the model. Lamping [72] distinguishes between the client interface of a class and its specialization interface. The former is used to access the functionality of objects of the class, while the latter is the interface between a class and the subclasses that specialize it. The authors suggest that current object-oriented languages do not properly capture the class dependency information implied by the specialization interface. By extending the application of the type system to permit classes to indicate how much of their state and functionality is relied on by each of their methods, it is shown how inappropriate utilization of inheritance can be avoided. The work by Lamping [72] seems to point out a flaw in Fisher and Mitchell’s observation [34] stating that if implementation modifications to an object’s specialization interface preserve its signatures, then inheritors need not update their code. In fact, inheritance is potentially dangerous since modifications to inherited methods that add functionality could modify the expected utilization of those methods by derived classes, though the “protected” interfaces have remained the same.

Madsen [81] presents a language mechanism enabling an object to change its virtual bindings at runtime is presented. It is shown that while there is currently a clean mapping of static class design into related programming language constructs, no such analog exists for the dynamic behavior of a class. Through the use of a replaceable vtable (virtual function table), it is demonstrated how state dependent behaviors can be easily modeled and implemented.
Before considering more drastic shifts in design philosophy and tool development, it is interesting to discuss the outerface mechanism presented by Harrison [59]. Outerfaces are used to combat the problems of component-structure fragility and missing-component fragility. Component-structure fragility occurs as a result of the code called by a function consumer being tied to a particular package of the supplier. This happens since the object implementing the interface is necessarily the one supporting the desired method. On the other hand, missing-component fragility is mainly a problem in weakly-typed systems where there is no guarantee that the method call being requested will be implemented by the runtime object. To solve both of these problems simultaneously, the outerface is used to express a statement of functionality without relation to its packaging. As described at the start of our discussion on object-oriented programming, outerfaces remove the special role of an object as a unique entity to which the message is sent. For outerfaces, the type-checking rule for method calls is that "the operation call being made must be included in the outerface of at least one of the parameters of the operation" [59:5] (essentially interfaces for multiple dispatch).

While on the topic of multiple dispatch, it is also useful to consider a more general approach called predicate dispatching. As stated by Ernst et al. [41:1], “predicate dispatching generalizes previous method dispatch mechanisms by permitting arbitrary predicates to control method applicability and by using logical implication between predicates as the overriding relationship.” To clarify the distinction, consider that single dispatch uses only the dynamic type of the receiver object to determine which method to call. In the case of multiple dispatch, the dynamic types of all the method parameters are used to make the determination. Finally, for predicate dispatch, arbitrary predicate
expressions are associated with each method definition thereby allowing not only the
dynamic types of the parameters but also additional features of the parameter objects to
be tested when making the selection among different method implementations.

Subject-oriented programming is a relative newcomer to language design
methodologies. It strives to be the next step in the progression from procedural, to
object-oriented, to component-based software development. The major players in the
component arena are Microsoft COM [90][102], OMG CORBA [102], and Sun’s
Enterprise Java Beans (EJB) [104]. Each of these technologies implements a different
approach to solving the same basic problems of cross-platform interoperability,
versioning, and language independence for the development of highly reusable objects.
Additional features such as transaction management, naming services, and resource
pooling are also usually added into the offerings. A significant distinction between
CORBA and COM is that the former permits implementation inheritance while the latter
does not. COM has chosen to use aggregation (delegation) as its form of differential
definition. Microsoft Corporation [90] defends this choice by pointing out that the
contract or relationship between components in an implementation hierarchy is not
clearly defined and is therefore ambiguous. While some readers may disagree, the author
further states that inheritance actually violates the principle of encapsulation as well.
Perhaps the techniques proposed by Lamping [72] could help to remedy these
shortcomings.

Returning to the issue of subject-oriented programming, we quote the seminal work by
Harrison and Ossher [95:412], where the overall goal of the model is specified as striving
to "facilitate the development and evolution of suites of cooperating applications."
Kaplan et al. [67] note that the techniques of inheritance and delegation in object-oriented systems are usually applied to the recombination of previously decomposed system models. The original decomposition takes place since it is easier for developers to tackle a large problem by breaking it into more manageable pieces. Kaplan et al., [67] adopt the view that recombination (or composition) of the decomposed modules is better performed through the techniques of subject-oriented programming than their object-oriented counterparts. Four common routes of decomposition are identified as possible design paradigms for superior application of subject-oriented techniques. As outlined by Kaplan et al. [67], these include:

1. Feature-based – a program to be developed is decomposed vertically into separate features (e.g., parser, editor, debugger, etc),
2. Staged – a horizontal decomposition occurs where minimal functionality is initially built and then additional functionality is added through composition,
3. Unanticipated Extension – after a complete system has been defined, new requirements arise – the new requirements are used to build a partial system that is then composed with the original, and
4. Unanticipated Composition – two separate systems are obtained and a bridge is developed to allow composition of the programs across their commonalities.

We see immediately that whereas the discussion of object-oriented programming is mainly geared towards the successful creation of a new application, subject-oriented programming is concerned with the maintenance and adaptability of existing programs. The first class entity has shifted from the object to the subject. This emphasis on the support of maintenance is substantiated by Wilma Osborne’s 1988 study at the National Bureau of Standards that finds 60% to 85% of the total cost of software is due to maintenance. Further support for these ideas can be found in Cox’s [26] and Winograd’s [131] work which suggest that the majority of time spent on object-oriented development
is not involved with creating new code but rather with integrating, modifying, or evolving existing code.

Before reviewing the specifics of subject-oriented programming theory, we highlight the problems it seeks to address. In general, subject-oriented programming can be used to create extensions to and configurations of software [64]

1. Without modifying the original source code.
2. Encapsulating deltas for multiple platforms, versions, and features.
3. Customizing and integrating systems and reusing components.
4. Facilitating multi-team system development.
5. Permitting decentralized development of classes while eliminating concurrency conflicts and centralization bottlenecks.
6. Maintaining correspondence between requirements and code.
7. Simplifying code for many design patterns through the use of unusual language features.

The primary tool for solving these problems is the “subject” in subject-oriented programming. As defined by Ossher and Harrison [95], a subject is a collection of state and behavior specifications reflecting a perception of the world at large. In general, a subject is manifested as a combination of a portion of the state and behavior of objects in many classes. To create useful applications, subjects are combined together using a number of different composition rules. The composition rules include, among other, information concerning interface matching and class matching. Interface matching is the term used to describe the act of determining appropriate correspondences among interfaces of composed subjects. Likewise, determining the correspondences between classes defined by composed subjects is called class matching. Using these techniques, subject-oriented programming can allow different subjects to impose their own
classification hierarchies over common objects. Subjects comprise portions of many objects and are then themselves combined into compositions. Each subject in a composition need only concern itself with the relevant portions of its underlying objects; however, the singular identity of the objects is still retained across the composed subjects.

In this way, an object developer need not face the impossible and counterproductive task of predetermining and implementing all intrinsic and extrinsic properties and behaviors of a particular class. "Even if a component producer could anticipate all possible uses a priori, the resulting interface would be too complicated, and most programmers would be overwhelmed by a multitude of methods, most of which they never need" [69:4]. As new properties are required, new classes can be developed, and the two classes can be composed into a subject carrying the full set of required information. As a side benefit, the problem of "object schizophrenia that results when the state and/or behavior of what is intended to appear as a single object are actually broken down into several objects (each of which has its own object identity)" is also avoided [64].

The Hyper/J [122] environment supports and extends the work on subject-oriented programming by providing the tools to perform composition and by implementing the theory of hyperspaces. As outlined by Tarr and Ossher, "hyperspaces permit the explicit identification of any dimensions of concerns of importance, at any stage of the software lifecycle; encapsulation of those concerns; identification and management of relationships among those concerns; and integration of concerns" [122:11]. In essence, concerns are similar to the decomposition paradigms presented above, and hyperspaces allow the developer to end the “tyranny of the dominant decomposition” [123:1] which
forces separation and encapsulation based on only a single concern (e.g., just features or just configurations).

Foote [35] suggests a number of linguistic support mechanisms for assisting in the development of subject-oriented programming. These include:

1. Dispatching Mechanisms – dynamic message routing and multiple dispatch,
2. State Mechanisms – version management, consistency maintenance, persistence, and constraint support,
3. Protocol Objects – allow designation of message groups for special treatment,
4. Namespace – control the visibility of different views of an object,
5. References – multiple subjects have different handles and therefore different views onto the same objects, and

Additional techniques to solve the same problems attacked by subject-oriented programming have also been proposed in the context of reflective object-oriented programming [82], perspectives and composite objects [119], and binary component adaptation [69]. Developers who only have access to the object code of the modules being composed [96] can apply the subject-oriented techniques of Ossher and Harrison [95] and Tarr and Ossher [122]. However, effective utilization of the composition rules is made easier through access to the source code. In the binary component adaptation (BCA) scheme proposed by Keller and Holzle [69], no access to source code is ever required. In fact, whereas Hyper/J creates its primary output at compile time, BCA performs its work dynamically at program load time.

As described by Keller and Holzle [69], BCA takes place after the component has been delivered to the programmer with modifications occurring in-place. Developers are
free to add new methods to a class, force two classes to extend some third class (to use
subtype substitutability), remove methods, add member variables, and perform other
operations. The advantages of BCA are that it requires no source code access, preserves
release to release compatibility, allows a wide range of modifications, can be deferred
until load time, and introduces only a small load-time overhead. Keller and Holzle
[69:16] claim that BCA can reduce "effort by enabling the reuser to more effectively
customize components to the needs of the particular application and by supporting
predictable and non-predictable component evolution."

Thus far, we have reviewed object-oriented programming, examined some of its
weaknesses, and investigated new technologies being used to enhance the development
experience. It should be apparent from the preceding discussion that much of the current
work is concerned with adaptability and maintainability of applications. In addition,
attention is also being paid to the interoperability and reconciliation requirements of
semi-independent, multi-enterprise development efforts [96]. An attempt at measuring a
software artifact’s dependency on type definitions can be found in the dependency metric
introduced by Lieberherr and Palsberg [77]. There, the authors attempt to engineer
adaptive software by separating a program into a generic, high-level, data-structure
independent algorithm to which some chosen data structure can later be mapped. A two-
stage compilation process, including class-library compilation and propagation, is used to
generate the final code. A similar attempt at excising the data specific aspects of a
program from the algorithms can be found in the work of Musser and Stepanov [91]
under the title of algorithm-oriented generic libraries.
2.3 Persistence Mechanisms

In certain application domains, data types may have complex and constantly evolving structure. For this reason, the rigid, static schemas prevalent in relational database designs are often inadequate. Semi-structured databases are rapidly being accepted as a solution to these issues. This view is confirmed by McHugh et al. [87:1], where the authors state that "semi-structured data can neither be stored nor queried in relational or object-oriented database management systems easily and efficiently." Since the eXtensible Markup Language (XML) is the most ubiquitous of the semi-structured languages, we begin with a brief review of the standard.

The W3C is the standardizing body behind the XML specification. Offered for public use to assist new users in learning the technology, the W3C has provided an excellent high-level document called “XML in 10 Points: (7, really…)” [7]. We briefly review these points here:

1. XML is a method for putting structured data in a text file.
2. XML looks like, but is not HTML.
3. XML is text, but it is not meant to be read.
4. XML is a family of technologies.
   a. XML 1.0 [10] is a specification defining what “tags” and “attributes” are.
   b. XLINK [83] describes a standard way to add hyperlinks to an XML file.
   c. XPath [23] describes a language for addressing parts of an XML document.
   d. Xpointer [83] based on XPath uses a URL-like syntax to point to pieces of data in an XML document.
   e. XML Fragments [54] define a way to send fragments of an XML document, "regardless of whether the fragments are
predetermined entities or not-- without having to send all of the containing document up to the part in question."

f. Cascading Style Sheets (CSS) [133] describes a style sheet language for adding style (fonts, color) to web documents.

g. Extensible Stylesheet Language (XSL) [1] is a more advanced language for expressing style.

h. XSL-Transformations (XSLT) [22] is a declarative transformation language for transforming XML documents into other XML documents.

i. Document Object Model (DOM) [132] is a standard set of function calls for manipulating XML files from a programming language.

j. XML Namespaces [12] provide a simple method for qualifying element and attribute names used in Extensible Markup Language documents by associating them with namespaces identified by URI references."

k. XML-Schema [118] is a replacement for Document Type Definitions (DTDs) and is used to place constraints on the content and structure of XML documents [126]. It is based on XML Data, Document Content Description (DCD), Schema for Object-Oriented XML (SOX) [30] and Document Definition Markup Language (DDML).

5. XML is verbose but that is not a problem. The XML tags take a great deal of space, but disk space is no longer at a premium, and compression of tags works very well.

6. XML is new, but not that new. Development began in 1996.

7. XML is license-free, platform-independent and well supported. [7]

In addition to the specifications provided by the W3C, non-W3C XML technologies also exist, such as the Simple API for XML (SAX) [88], which is a standard interface for event-based XML parsing. With the overabundance of specifications, it would seem that XML would be difficult to approach. In reality, it is a very straightforward language that can be used to easily accomplish a number of otherwise difficult tasks.
Bartels [5], of POET Software Corporation, an object-oriented database company, details the key requirements for an XML repository. These include:

1. A flexible storage model to support changes to the underlying XML-Schema.
2. A developer view exposing only native XML.
3. Support for XLink including bi-directional links.
4. Legacy data support for accessing external non-XML data sources.
5. Component model to prevent impedance mismatch between object-oriented languages and the persistence format.
7. Scalability to handle large volumes of data and simultaneous transaction-based access.
8. A well-defined API for programmatic extraction of needed information.
9. Easy to use, powerful database administration tools including security, deployment, and backups.

The use of an object-oriented database management system (OODBMS) for storage and retrieval of semi-structured data is most certainly backed by POET Software Corporation, but it also seems to be acknowledged by the Lore team at Stanford [86]. Lore is a semi-structured DBMS based on Object Exchange Model (OEM). Lore differs most significantly from its pure OODBMS cousins in that it provides no fixed schema. Rather, a DataGuide is provided as "a concise, accurate structural summary of a semistructured database" [50:1]. The DataGuide has proven to be valuable for browsing, query formulation, storing statistics, query optimization, and compression. The basic approach to query processing in Lore is to parse the query, convert the parse tree into an OQL-like query, construct the query plan, perform optimizations, and then execute the
resultant plan [86]. The techniques for execution have been broken down into four main categories.

In top-down query execution, path expressions are executed in a forward manner starting from the top of the directed graph representing the data. This technique accumulates its result using a recursive iterator approach, where each node in the query plan requests a node at a time from its children. To enable utilization of indexes, the Lore team also describes a bottom-up approach to query execution. Here, objects satisfying a where clause are first isolated at the leaves of the graph and then propagated upwards toward the root. Identification of the desired leaves is performed through the use of an index. A hybrid technique can also be applied where top-down and bottom-up executions meet somewhere in the middle. Finally, an inside-out query execution strategy identifies "portions of the graph that match the middle of a path expression, then traverse[ing] up through the parents and down through the children to match the remaining portions of the path expression" [87:6]. To accommodate these various query execution strategies, a number of novel indexing techniques were also created. A value index (Vindex) is used to locate atomic objects with certain values. A text index (Tindex) is used to locate words or groups of words. A link index (Lindex) helps locate the parents of a given object. And finally, a path index (Pindex) provides fast access to all objects reachable through a given labeled path.

While Lore is not the only semi-structured database available for real-world implementations, it is an excellent example of such a system. If space allowed, a more detailed description of Lore’s feature set would show that it addresses most, if not all, of the repository qualities emphasized by Bartels [5]. Another interesting form of semi-
structured storage provides for the inclusion of behaviors. For instance, in OOXLM [110], behaviors are stored in XML as LISP lambda expressions. These behaviors accompany their associated data and provide the processing context for handling new data types. This two-language approach to software development has been suggested before. For example, Masse [84] discusses the advantages obtained from mixing the static type checking of Java with the dynamic types of Python. To the best of our knowledge, the use of XML as a full-fledged external type system for an accompanying programming language is a mostly unexplored area of research; however, it is one we believe will lead to novel solutions in the C2B domain.
An initial stumbling block for C2B technologies is the inability to cope with incomplete and ever-changing requirements. This problem is most apparent in the context of complex, dynamic, and poorly specified domain-specific data types. These data types are rarely solidified at the time of implementation, and, even more troubling, they are very likely to change over the lifetime of the application. A successful system must be able to cope with such occurrences without overburdening either developers or users. A further problem arises as a result of the inherent complexity of the data types involved in many business transactions. From a global perspective, the complexity is a real requirement. However, it is likely that only a small subset of that complexity will be needed to accommodate any single behavior. Forcing the full instantiation of a highly complex object is not only inefficient, but it may also be impossible if some of the required data is unavailable.

The ability to represent complex, dynamic, and poorly specified data is only the first challenge. An equally important and difficult problem arises in the communication of such data. The traditional programming paradigm is to define data types that can be communicated effectively and efficiently throughout a single application. A more pragmatic and forward-looking approach is to give initial consideration to the enterprise as a whole. This recent step has produced admirable results, but still falls short of our present needs. Consumer data, and also business data for that matter, must be transferable across enterprise boundaries. More difficult than just inter-application
communication, this requirement seems to force some agreement or standardization mechanism between parties. Unfortunately, such large-scale cooperative efforts often prove very difficult to implement in the real world. So, while the requirement for global communication still stands, we must somehow relax our intention for global reconciliation and compromise through the use of an alternate methodology or semantic interpretation.

Finding suitable representation and communication techniques, brings us only halfway towards our goal of a viable C2B solution. The next challenge concerns persistence of the new, complex data representations. Once again, we must expand our view outward past the enterprise-centered systems of today. The importance of maintaining the singular identity of a persisted object, regardless of where or how frequently it is written, cannot be understated. Across enterprise boundaries, and even within the myriad systems of a single enterprise, it is usually very challenging to determine and guarantee the physical equality of two persisted entities. This shortcoming must be resolved if we are to succeed in creating a truly ubiquitous type definition, one whose instantiations are always capable of resurrecting their unique identity. To further complicate the matter, we perceive an equally important requirement to be the maintenance of physical equality though the persisted elements of the object may be entirely disjoint. The ability to reconcile scattered notations of an entity’s attributes, and realize them as a single individual, should frequently prove invaluable in a world of ever changing needs and unpredicted utilization. Finally, the support for fast, efficient queries and simple, easy to implement maintenance protocols against potentially massive stores of non-identically
structured entities, closes out the persistence requirements for our newly envisioned system.

In a move that goes against current thinking in software design methodologies, we add our final provision. Even to the extent that the underlying data type would require modification, we feel that the instantiated entity must be modifiable by the addition, removal, or mutation of its attributes and behaviors. Furthermore, regardless of how a given entity is modified it still must be possible to communicate it and persist it along with its unique identity. This requirement emphasizes our belief that representation of whatever is deemed important far outweighs the benefits of a static data type specification. For instance, given a class that represents a person’s features, if an application needed to store a new eye color attribute it should be able to do so even though the underlying class definition has no notion of eye color. This behavior is provided in the JavaScript language [92] where the members of a class are stored as elements of an associative array.

In describing the properties important to our envisioned C2B system, we have highlighted the attributes of a broader class of problems. The exposition introduced a number of new and diverse ideas, so we now follow with a summary outline to help clarify the important points and avoid any misconceptions.

1. Data types are complex, dynamic, and not well understood at the time of implementation.
2. Communication of entities across enterprise boundaries is imperative.
3. Persistence of entities including their unique physical identity must be possible.
4. Different views of an entity must be persistable without loss of identity.
5. Large stores of persisted entities must be fast and easy to query and maintain.

6. An instance of an entity is modifiable even to the extent that the underlying data type must change.

3.1 Research Goals

The primary goal of our research is to develop the tools and techniques required to realize a functional and practical C2B infrastructure. Our intention is not to develop a marketable system, but instead to envision and specify the required features for such a system.

To help uncover our sub goals, we examine a number of use cases for the hypothesized system. Please keep in mind that the scenarios presented are not requirements for an implementation. Instead, they are modules that could be developed based on our proposed architecture. Further on in this dissertation, we provide a description of this architecture and more carefully examine each of its components. For the present, consider the few scenarios presented as a representative sampling of the flexibility we hope to obtain.

3.1.1 Mobile Shopper

A pedestrian is taking a walk through the streets of San Francisco. She passes by many shops and frequently pauses to peer through the windows and examine the various offerings. With so many stores to see, she rarely enters one unless something quite special catches her eye. But today, something different is about to happen. A clothing store, a short distance down the street from her current location, recently installed a new C2B system. Sensing the young woman, the software requests her identity, and based on it determines her clothing preferences. Reviewing current inventory, the software finds that the shop stocks a number of items that seem to fit her profile. The pedestrian is
notified of a special sale on these particular items by having a message pop-up on her personal digital assistant (PDA). She is given directions to the store from her current location, and soon after becomes a new customer.

3.1.2 Emergency Room

A businessman has just fallen ill after a dinner with some potential clientele in New York City. He is far away from his home and his family physician in California. The ambulance transports the gentleman from his hotel room to the emergency room at the nearest hospital. As a result of the food poisoning, the patient is in no condition to fill out insurance forms or to answer questions regarding family history or allergies to various medications. But today, nobody even asks. His identity is established, and authorized hospital personnel retrieve his medical information. In addition, the family doctor on the other coast is notified of his patient’s condition. The local doctors prescribe the necessary treatment, and the following morning our businessman is recovering without complications.

3.1.3 Job Searching

A generation X’er has tired of her job at the high-tech firm of ComDotCom. Having worked for 11 months, it is certainly time to find another job. Her last employment change required a state-to-state move, ten interviews, and an unendingly repetitious trail of paperwork. Each interviewer required that she complete the same set of forms, and in just the same way, so did each lease application. But this job change will be different. The young woman advertises her intent on the Internet, and as expected, the interviewers come calling. She lines up the companies of interest and permits them to access the data required to complete their paperwork. With her new location established, she expresses her interest in an apartment equivalent with her present dwelling, and soon after, is
presented with a number of reasonable options. She isolates a few interesting prospects, and the lease applications are filled automatically. A few weeks later she is happily employed at DotComDot.

3.1.4 Insuring The Family

For the Hendersons, the cost of car insurance just seems to keep increasing. Their youngest son has just received his permit, and the rates have soared to new heights. With both parents working, there is really no time to deal with the hassle of calling twenty different insurance companies to find the best rate. But today will be different. Wanting a number of changes to their coverage, the Hendersons modify the description of their current policy. The description is not immutable and is created to include a number of equally acceptable alternatives. The utility function representing their requirements is sent onto the Internet and eventually matched with a willing insurer. Using public keys and digital signatures, the Hendersons are enrolled with the new policy and receive their insurance cards the following morning.

3.1.5 Proactive Treatment

John Smith, a sufferer of acute somethingitis has always tried very hard to keep abreast of the latest medical treatments available for his condition; however, such treatments are few and far between, and often he will lose touch for years at a time. He has played catch-up many times in the past, but that is about to change. Anonymously placing his relevant medical information on the Internet, John expresses interest in receiving treatments relevant to his profile. Many months pass, with our patient predictably having forgotten about his search. However, this morning a small pharmaceuticals company has just released a new applicable treatment. The company initiates a search for patients with matching profiles and finds John. The following day
he receives a notification to his anonymous mailbox and soon after his condition is improved from the new treatment.

3.1.6 Large Scale of Software Design

A large multinational company wishes to develop a highly extensible software system. The proposed system is composed of many distinct applications and can also be further extended by customers to suit their own proprietary needs. In addition, the data requirements for the system shift frequently with the seasonal product changes. In fact, it is nearly impossible to define any fixed type definitions that is of use to all involved applications. The software engineers look at the product requirements and shake their heads, but one of them looks up. He knows that the theory underlying a certain C2B architecture is directly applicable.

3.1.7 Outline of Goals

Having outlined a number of conceivable use cases we are now in a position to abstract the commonalities and produce a working list of goals. These goals, generic in their expression, strive to convey the practical applications solved by our proposed architecture and theory:

1. Support mobile/non-mobile users in obtaining real-time, highly relevant information and personalized attention.
2. Support mobile/non-mobile users in the maintenance and access of their personal data for a variety of real-life situations.
3. Enable creation of semantically enriched “forms” for automated information extraction/completion from personal information sources.
4. Enable matchmaking between personal requirements and corporate offerings.
5. Enable automated update of personal information at the completion of specified transactions.
6. Enable registration of interest through profile description along with means for anonymous notification of discovered matches.

7. Enable system creation in the absence of well-specified type definitions or well-formulated requirements.

8. Enable automated system evolution in support of new, custom data.

3.2 Additional Details

The goals provided above give us a clear direction and work well to motivate the definition of our objectives. While it is tempting to approach this topic purely from an implementation perspective, it would certainly be unwise. The goals proposed might appear rather innocuous on the surface, but as implementation proceeds, the weaknesses in the current software development tools and techniques quickly become apparent. Let us consider a quick example to isolate one of the major weaknesses in modern software design.

Assume we want to develop a program that can instantiate a never before seen class at runtime and begin referencing its methods. Further, let us assume that the data for the arguments of the methods, as well as the class name itself, are provided in an XML document. When we parse the XML document, we can find the name of the class the user wants instantiated and use the class loader to obtain a Class of this type. We can make a simplifying assumption that the class implements a well-known interface. If we are not going to use the default constructor, then we must also have some means of determining which constructor to use. This can be done relatively easily with reflection if we know the signature of the constructor for which we are looking. To continue, let us say that we find the constructor and use it to instantiate the new object with whatever values the user placed in the aforementioned XML document. With the newly created object, we are now in a position to begin calling its methods. These can also be
discovered with reflection, or we may know them beforehand based on the implemented interface. So, where is the weakness?

We have provided an example that uses the latest technologies to produce a very dynamic and extensible application. However, the end result is still dependent on our prior knowledge of the semantics surrounding the instantiated class and each of its member methods. Though we can use reflection to find and call the methods, we have no apparent means of determining the semantics underlying their signatures. That is, the fourth argument of the ZZZ() method of class Sleepy has no discernable relationship to the real world. If we did not know about that method before we started, then it would require some very fancy footwork to make sense of it. And, it is that fancy footwork we hope to simplify.

The design example discussed above highlighted a very interesting scenario to which any number of solutions may be applied. We could insist that all method names follow a given convention, we could use an XML document as a script and put the method arguments in some specified order, or we could create some kind of ontological specification for all of our interesting methods. Each of these techniques would produce useful results, and each would also require a substantial amount of effort. But even if that effort were expended, we would still not have created a solution to meet all of our goals. Consider now that the data handled by the classes and methods frequently changes, and that many people all over the world depend on its constant availability. If one person makes a change, where would that leave everybody else (remember stretchability)? Factor in to these problems the issues of inter-application communication, data persistence, mobility, matchmaking, and automated update propagation, and it
immediately becomes apparent that the area of C2B is replete with interesting and
challenging computer science research topics. With the transformation from business
requirement, to implementation problem, to research topic complete, let us take a look at
our objectives:

1. Describe a high-level architecture that will act as the infrastructure and basis for
   the remainder of the research.

2. Hypothesize a set of properties required of future software engineering processes
   and software development tools that will alleviate the deficiencies in the present
   offerings.

3. Further refine the concept of stretchability, perhaps providing a quantitative
   interpretation based on the binding time of a program’s variables and type
   definitions.

4. Devise new agent-oriented design concepts and new language features such as
   XML method signatures and global type definition repositories.

5. Document new design methodologies and implement the proposed architecture
   and new language features.

6. Create a working implementation of a C2B system implementing one or more of
   the use cases outlined above.
CHAPTER 4
STRETCHABILITY

In this chapter, we introduce a new software quality measure called stretchability. The reason for this is twofold. First, a firm definition of the term aids greatly in the expression of our intentions. Second, its properties as a metric hopefully allow us to quantify some of the notions we propose.

The motivation for creating this measure stems from a need to understand the requirements of Consumer-To-Business (C2B) systems. Consumer-to-Business (C2B) is really a combination of direct marketing and self-marketing. Direct marketing is not a new idea; many companies are already actively engaged in finding ways to deliver personalized messages to their customers. More than just the personalization of patrons seeing their own names, customization strives to invoke statistics and past experience to better predict and serve a customer’s future needs. Whether proactively thinking on the consumer’s behalf or simply providing a more productive and engaging purchasing experience, direct marketing is a key ingredient to increasing customer loyalty and, consequently, profits.

On the other side of the equation, and perhaps more aptly labeled C2B, self-marketing is the intentional submission of consumer information by the consumer to the corporation. Here, for the first time, the consumer is the motivating entity behind the marketing process. No longer is personal information garnered for the purpose of subjecting the potential client to unfiltered and undesired solicitations; now the consumer
tells the business what he wants it to see, and the business is freed from the expense and burden of thousands of unrequited advertisements.

Though the excitement and richness of new ideas generated by the concept of C2B is hard to resist, as computer science researchers, we must discipline ourselves and look at the related problem of semantic self-adaptability. The data types involved in the proposed C2B transactions are likely to be as volatile as the products and services they represent. Any software designed to support such a domain must be able to retain its utility independent of the frequency of semantic modifications its fundamental data types may have undergone. That is, the software must be able to adapt to the changes in the semantics of its application domain. The concept of stretchability is introduced to provide a quantifiable measure of the semantic adaptability of a given piece of software.

Stretchability is best defined as an internal software quality akin to evolvability and reusability but focused primarily on type definitions rather than whole systems or modules. Evolvability is a property attributed to systems designed to enable the addition of new functionality with minimal effort. Reusability is a software quality that refers to systems whose modules or components may be reused with minimal modification to help create an entirely new system. Stretchability refers to a system’s capacity to absorb changes to underlying type definitions and to accept entities of unbeknownst type definition from external sources. Since the evolution of a system often requires modifications to type definitions, it is clear that stretchability assists in evolvability. Furthermore, since many systems differ little except in their utilization of different domain data, stretchability can be considered a support mechanism for reusability.

Finally, and very importantly, it should be noted that stretchability clearly differs from
other software qualities in that it involves immediate consideration of external sources. This clearly reflects the intent and importance of this metric as a measure of the suitability of designs and implementations of ubiquitous systems. In closing, it should be mentioned that stretchability is an attribute that can be applied with equal import to process as well as product.

It is our belief that future systems must exhibit ever-increasing degrees of stretchability to succeed in a ubiquitous environment. These new systems will be developed under the constraints of incomplete and unstable requirements and will enable the creation, communication, and externalization of data along with completely supportive semantic interpretations.

In contrast to the intended application of other metrics, such as lines-of-code or mean time between failure, stretchability does not seek to uncover a quantitative attribute. Instead, this new measure focuses on describing inherent properties of software that simplify and support cross-enterprise reuse, maintenance, and communication. When discussing stretchability, it is vital that we view software as a process consisting of many tasks and artifacts rather than simply the delivered version of the executable. It is true that the average software user has no concern for these development processes or additional artifacts; in fact, we would argue that the tools and techniques used to design and implement software are primarily convenient mechanisms for the developers and should not be relevant or even discernable to the end users. However, the fact remains that good software has a long lifetime, and the majority of that time is spent in maintenance or realized as reuse. Continually satisfying the user base requires frequent evolution, and this is where development techniques begin to have a direct impact.
4.1 Introduction

Software engineering literature is filled with case studies and suggested techniques for producing enterprise level systems. Preference has shifted from the original waterfall model [47] to the now popular spiral lifecycle [6], but the emphasis always remains on bounded system development. A bounded system has reasonably well defined requirements and is developed to achieve known goals. Furthermore, a single team, or a number of teams with well-established communication protocols usually develops such systems. This enterprise-centric view of development leads to improved system designs, but it does not scale well to the distributed cross-enterprise systems we envision in the future. To achieve those goals, we must attend to the presently limiting definition of a system. We can no longer consider a system to be the result of a large-scale implementation effort by a single enterprise, nor can we simply qualify this definition to require the presence of many cooperating enterprises. In both cases, we victimize the potential for autonomous development.

As an example, consider a software project with the ultimate goal of enabling consumers to find the best price on an insurance policy. Consider further that the project is not controlled and initiated by a single centralized authority but is instead driven by consumer interest. Consumers want the ability to compare different automobile insurance policies from various companies. However, the consumers do not want to re-enter their data at each insurer’s web site. Rather, they want to place their data on the Internet, have the insurance companies discover that data, and offer their best possible price. To complicate matters even more, the insurance companies do not want to collaborate with each other to produce a standard insurance submission form. In
addition, the specific data elements required by each insurer differ with respect to the liability class insured by each company.

Given this scenario, it should be clear that the present state-of-the-art in software design methodologies and software development tools are not sufficient to fully achieve all of the desired goals. For instance, the lack of a central authority to help define the format of the insurance information submitted by consumers will almost certainly lead to insurmountable integration problems. To clarify the need for integration, consider a common query that may be posed by a consumer: “Please find me the insurance company that will write the cheapest policy with equivalent coverage to my existing policy.” Two primary events must occur for this question to be properly answered. First, the current insurance company must implement a means to export a consumer’s policy description. Second, another completely independent insurer must be able to import the policy description and determine the best price it can provide on an equivalent policy. Even if we ignored the problems of initiating data transfer between the two systems, we would still be left with the semantic mismatch between the policy representations used by both insurers. The concept of stretchability is introduced to help solve the problems with this semantic mismatch, as well as the more mundane issue of reliable information transfer between autonomous systems. The remainder of this dissertation is dedicated to clarifying the domain of stretchability, as well as the properties it implies in conforming systems.

4.2 Current Technologies

We have introduced autonomous development as an important ingredient in achieving stretchability, but it is not sufficient in and of itself. For the remainder of this dissertation, we use the term entity to refer to an individual, a small business, or an
enterprise. Essentially, an entity is any individual or group of individuals that develops or uses software. Consider the interesting, though perhaps constraining, technique of quantifying the notion of stretchability by tallying the number of autonomous entities capable of developing inter-workable components. Based on this definition, it would be tempting to view the entirety of COM objects [8] or Enterprise Java Beans (EJBs) [3] as composing a stretchable system. These technologies are certainly a step in the right direction, but they still lack the inherent semantic compatibility required for true interoperation. More specifically, stretchability has not been achieved when the method signatures of some previously unknown interface are successfully parsed; the capability to call the methods is empty in the absence of understanding what functionality those methods perform.

That said, it would be misleading to imply that there has not already been some progress in the direction of stretchability. Standards, such as ODBC or JDBC [3] provide a layer of insulation between applications and relational databases. Further abstraction can be achieved using interfaces, such as ActiveX Data Objects (ADO) [121] where the data sources can include flat files, mail, video, and mainframe and legacy data in addition to relational stores. Technologies, such as COM (DCOM) [53] and CORBA [102] helped to achieve a degree of language independence and helped to make process boundaries and machine boundaries more transparent. Microsoft’s new .NET [99] initiative promises to further advance the notion of language independence enabling such possibilities as deriving a Visual Basic class from a C# class which is in turn derived from a C++ class. The primary enabler is the Microsoft Common Language Runtime (CLR) that is essentially a virtual machine able to interpret a new byte code format. Any
language that compiles down to this intermediate language can take advantage of the features provided by the environment. Microsoft has chosen to use this to implement language independence, whereas Java used the same ideas to promote platform independence.

Standards, such as the Lightweight Directory Access Protocol (LDAP) [3] and the Java Naming and Directory Interface (JNDI) [3] have helped to hide the details of unique resource naming, discovery, and access. Other standards, such as COM+ [53] (formerly Microsoft Transaction Service (MTS) [70] and now part of .NET), or Enterprise Java Beans (EJB) [3], Java Transaction API (JTA) [3] and Java Transaction Services (JTS) [3] simplify the implementation of distributed transactions by enabling standardized components to easily enlist themselves in existing transactions or to start new transactions for themselves. Finally, technologies such as Microsoft Message Queue (MSMQ) [70] and Java Messaging Service (JMS) [3] can assist applications in achieving reliable messaging.

All of the technologies mentioned above, and many more not mentioned, are helping application developers to produce larger, more complex, and more scalable applications. However, the primary focus of these new technologies is to simplify the development of enterprise-level systems. This is an important step, but it is not the intended domain of stretchability. Enterprise-level systems strive to coalesce the knowledge of a multi-site, multi-department corporation into a single cohesive unit. Intranets are often built between national offices, and the global availability of the Internet is leveraged between international offices. With the communication channels intact, the new enterprise applications can share data and processes enabling the development of corporate data
warehouses and application server farms. The new found cohesiveness may assist in realizing a competitive edge for the corporation, but it does less to help achieve the cross-enterprise requirements of supply-chain management or the consumer-centric goals of direct marketing and self-marketing.

The need for cross-enterprise supportive technologies has not been overlooked. Standards such as Electronic Data Interchange (EDI) have existed to serve this purpose for many years. Recent developments based on the eXtensible Markup Language (XML) [11] are also leading to new solutions for these problems. One example of such a system is Microsoft’s BizTalk [76] framework, which utilizes the Simple Object Access Protocol (SOAP) [9] for transferring XML remote procedure calls (RPCs) over the HyperText Transfer Protocol (HTTP). In the same way that Java Virtual Machines (JVMs) and Microsoft’s CLR provide the potential for programming language and platform independence, these new technologies leverage a common ontology to provide semantic independence. In the case of EDI, the standard is itself the ontology, whereas in BizTalk, a framework is provided into which enterprises will eventually place their own ontologies.

Moving even closer to the goals of stretchability, are a handful of recently introduced technologies and specifications. The Web Services Description Language (WSDL) [20] is described “as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services)” [20]. Moving even further towards cross-enterprise integration, “the Universal
Description, Discovery and Integration (UDDI) specification … is a sweeping initiative that creates a global, platform-independent, open framework to enable businesses to (1) discover each other, (2) define how they interact over the Internet, and (3) share information in a global registry that will more rapidly accelerate the global adoption of B2B eComerce” [127:3]. Finally, the Electronic Business Extensible Markup Language (ebXML) initiative defines their goal as providing “an XML-based open technical framework to enable XML to be utilized in a consistent and uniform manner for the exchange of electronic business (eb) data in application to application, application to human, and human to application environments – thus creating a single global electronic market™” [39:7].

As systems begin to utilize these new technologies to a greater extent, they simultaneously begin to exhibit a greater degree of stretchability. Providing a definition for stretchability will assist in understanding what these technologies are trying to achieve and also provide us with a basis for comparison of different solutions. Finally, and perhaps most importantly, a clear definition of stretchability will help elucidate the issues involved in creating cross-enterprise software and ensure that all problem areas are identified and eventually researched.

### 4.3 System Definitions

The ability to enable enterprises, or any purposeful entities, to autonomously develop semantically interoperable components is the primary motivator behind the introduction of stretchability. A stretchable system is one that achieves this goal, as well as many others. Before continuing to introduce these additional goals, it is important to clarify the different roles played by the various components of a stretchable architecture.
4.3.1 Global System

A global system is not developed by any single entity; it is instead, a conglomeration of the efforts of many autonomously functioning entities and their implicit interactions. Furthermore, a global system has no formalized or fixed requirements outside of supporting stretchability. We simply do not know, or even care, about the details of the specific components used to implement the system; the only goal is to enable them to function together. To help clarify these exact requirements, we introduce a new software specification model called the 6FR in Section 4.4.

4.3.2 Constituent Systems

The second type of stretchable system, a constituent system, could be considered nothing more than a set of components housed within the framework of the global system. This definition, however, limits the intended scope of a constituent system and identifies nothing about the owning entities. In general, a constituent system is written by a single entity. If multiple entities are involved in the development effort, it is assumed they share close communication and common goals. The system itself will be composed of many components (modules), only a subset of which will interact outside the context of the constituent system (with other components of the global system). The remaining components will serve to provide the functionality we presently associate with enterprise-based software. Thus, a constituent system really has two separate sets of requirements. The first set defines the business (or personal) needs of the owner, while the second defines the level of stretchability supported to enable interaction within the global system. Somewhere in the middle there may also be a translation layer used to marshal information in and out.
4.3.3 Global Framework: Clarifying the Global System

The concept of a global system deserves additional attention. It is essentially a fundamental shift in the way we define completion of a development effort. A good analogy is the birth of a company. When a new company is started, the founders do not assume they have achieved success simply because they have paid the rent, hired employees, and developed a product. Their efforts are as yet incomplete. After product development, marketing efforts (beginning with advertising and the dissemination of a strong sales force) promise to expand visibility into the global marketplace. Transactions for company resources are not forced to occur on site. Rather, the salespeople proactively explore external opportunities and initiate and complete exchanges wherever they find customers. Essentially, the business is performed outside of the company, and the results are reported back. In the analogy, the global marketplace is performing the same role as our global system. Both expose well-defined frameworks into which new companies, or new constituent systems, can be introduced, and both provide a translation medium to help move business requirements from one enterprise to another. From now on, when an enterprise decides to undertake a new development project, it can no longer claim completion when the code is written. Instead, it is at that point that it must begin to expand visibility into the global system. It should be noted that these facts only hold true in the context of constituent systems. Isolated (bounded) systems do not require stretchability and are tangential to the scope of this work.

A question arises concerning the existence of a global system in the absence of any constituent systems. In such a case, we are left with an empty global system but one that is capable of accepting new constituent systems at any time. The footprint, or architecture, of the empty global system is another area of potential confusion. To
address this problem, a distinction must be made between a global system and a global framework. Without delving too deeply into the issue of design, we can define a global framework as the standards documentation, supporting object libraries, communication media, and globally available standardized executables required to enable development of a constituent system and enrollment of a constituent system into the global system. It should be made clear that the specification of a global framework also includes standards to be followed when developing constituent systems. Thus, a global system can now be defined as a global framework and whatever constituent systems are presently active within that framework. Given these definitions, it can be seen that two distinct development efforts are required. First, a minimal implementation of a global framework must be created and made publicly available. The actual implementation effort can be assumed by the same party that defined the global framework (as with Microsoft COM) or by third parties using a specification resulting from the framework definition (as with CORBA). The second implementation effort is ongoing and involves the construction and enrollment of constituent systems by an unlimited number of autonomous entities. The constituent systems must, of course, be developed in conformance to any rules specified in the definition of the global framework.

There may be many ways to realize the goal of a true global system, but one of the most direct will be to develop an extensible architecture supporting communication, transactions, replication, and some means of global semantic reconciliation. The ideal design would not require any a priori interaction between the constituent system developers to derive the full benefits from the global system.
4.3.4 Granularity

The general view of a system suggests that it is composed of components and modules. Unfortunately, the term component has become almost synonymous with a binary Microsoft COM object [8] or Sun Enterprise Java Bean (EJB) [3]. There are certainly other definitions, but the notion of a component as a distributable object exposing a well-defined interface is the most widely accepted; and Microsoft’s version is likely the most widely implemented (though by some counts, EJB has now taken the lead in new development efforts). The term module is usually used to refer to a cohesive piece of code that performs a well-defined task in a larger overall system. Given these two definitions, it is not unreasonable to use the terms interchangeably. Perhaps, it would be acceptable to define a component as a binary distributable object composed of one or more well defined code sections called modules, but then modules only appear in the context of components and that is obviously incorrect. For this reason, it is probably best to keep the two on equal footing and state that a module can be composed of one or more components, and likewise, a component can be composed of one or more modules. Furthermore, we can choose the term component when referring to the binary distributable version of the functionality, and the term module when discussing the bounded sections of code that implement the functionality. An agent is essentially a component that may or may not support mobility, autonomy, and a mental state.

The issue of granularity arises when we try to pigeonhole a constituent system into a predefined category. A single constituent system is most likely composed of many interdependent modules. As described earlier, only a subset of those modules will ever interact outside the context of the constituent system; that is, within the context of the global system. The requirements for the globally exposed components will be very
different than those for the isolated components. In particular, the exposed components will conform to the operability requirements of the global framework. We will refer to the globally exposed components of a constituent system as agents. This convention fits with our earlier analogy to the global marketplace, since a constituent system’s agents provide much the same functionality as the sales or support agents of a corporation. Furthermore, we can now attempt to categorize the individual agents of a constituent system into a particular classification, rather than try to compartmentalize the entire system itself. If this level of granularity still fails to provide enough separation in terms of functionality and utilization, then either the agent’s definitions need to be refined, or a further decomposition of particular agents needs to be undertaken. A shared component (agent) used within (by) many agents may be a good candidate for separate requirements, and implies an increase in the level of granularity.

4.4 Six-Faceted Requirements

The first step in the design of any software system should be the formulation of a precise problem definition. A valuable artifact of this stage is a high-level vision statement that helps to guide the remainder of the process. Once the team has a reasonable idea of what they are trying to accomplish, they should next refine their ideas through a rigorous series of requirements-gathering exercises. Examples of such exercises include brainstorming, storyboarding, questionnaires, prototyping, use-case analyses, and interviews with potential users. Assuming that the problem truly benefits from a software-based solution, the primary artifact of this stage should be a software requirements specification (SRS), which among other things, includes a numbered list of the software requirements. The particular form of each requirement is a topic of much debate, but there seems to be consensus that each requirement should provide the user
with some definable value in the context of the intended utilization. Since the birth of new software systems begin with the creation of an SRS, we feel that the definition of stretchability is best introduced at this same level. To assist in this endeavor, we propose a six-faceted requirement (6FR) structure in which the general requirements for the system are subjected to the questions of who, what, where, when, why, and how.

The 6FR enables us to establish a structured format for describing high-level component features. Whereas a language like Z [117] strives to formalize specifications through first-order predicate logic and Zermelo-Fraenkel set theory, we take a more pragmatic approach and try only to provide a well-defined requirement structure into which natural language answers can be placed. However, to make the natural language more precise and consequently, the requirements less ambiguous, we provide a general vocabulary based on the definition of low-level primitives called software specification axioms (SSAs). In reality, the comparison between Z and the 6FR is somewhat misleading. Z creates a formal model of all requirements within the system, while a 6FR tries to pigeonhole an overall system, or its high-level subcomponents, into a definable category.

An SSA affords its user a simple, well-defined, and reusable concept on top of which the answers to each facet of a 6FR can be built. After introducing a number, but by no means an exhaustive list, of SSAs, we introduce the notion of requirements patterns (RPs). Finally, we define the stretchability metric in terms of requirements patterns. To help clarify and situate the definitions, we explore a bi-directional relationship in which RPs imply stretchable systems and stretchable systems imply certain RPs.
Before continuing with the definition of the SSAs, we construct a more precise definition of a six-faceted requirement. As stated above, each such requirement is subjected to the six questions: who, what, where, when, why, and how. It is also important to keep in mind that we are analyzing the general requirements for the highest-level components or modules of a large system. The following list outlines the intended purpose of each of these questions. The remainder of this section defines the basic primitives used to answer the questions.

1. **What?** What is the requirement? This is perhaps the most fundamental question. The answer provided should be a short, well-defined statement providing enough detail to avoid ambiguity, and at the same time, avoiding excessive detail that may unnecessarily constrain design.

2. **Who?** Who are the primary users of this system component? For our purposes, the level of detail provided when answering this question should not initially uncover individual actors such as John the CFO, the marketing department, or the external stock quote feed. Instead, the goal is to generically identify which enterprises are responsible for maintenance, which are capable of maintenance, and which are affected by maintenance.

3. **Where?** Where will this feature of the system ultimately be deployed? In our earlier description of the stretchability metric, we highlighted its relationship to cross-enterprise software. While not limited to cross-enterprise scenarios, it is shown that there is a direct relationship between where a system is deployed and how stretchable it needs to be.

4. **When?** During what time interval is the requirement on the component valid? As the very natural result of system evolution, a large number of software requirements eventually undergo modification. Providing even an estimated answer for this question can alert the designers and developers to portions of the system that need to be implemented to support frequent modifications. Another important dimension of this facet identifies what changes can occur and how frequently those changes may happen.
5. Why? Why is this component in the system? Understanding the underlying impetus for inclusion of a particular feature permits a more precise understanding of its resiliency in the face of change. This facet is not used in the context of this dissertation, because our present goal is defining stretchability in general, rather than the requirements of some particular system.

6. How? How is this feature be implemented? This final facet diverges into the realm of design. However, it should be noted that in certain situations, particular implementation details might be legitimate requirements (for instance, the choice of a programming language dictated by the standards of a government or corporation). As in the case of “Why,” this facet will not be discussed further.

4.5 Software Specification Axioms

With the initial structure and definition of a 6FR more clearly defined, we are now prepared to introduce a number of SSAs. Recalling that the primary purpose of an SSA is to help answer the questions posed in a 6FR, we proceed by tackling those questions one by one.

4.5.1 Facet “What”

We begin by trying to define what purpose a component plays in the overall system. With the potentially infinite set of possible components, this may seem an untenable task. However, by abstracting frequently occurring commonalities, we find a number of concepts that can be reused, or instantiated, in the context of more specific problem domains.

4.5.1.1 Communication intent

It is possible to categorize the intent of inter-agent communication into three major types: data transfer, content transfer, and knowledge transfer. Data transfer is the act of moving data from one agent to another without any prior contract between the sender and receiver regarding the content or meaning of the transferred data. This type of communication is very simple but can still be useful for such tasks as database loading, application logging, and file transfer protocols. It should be immediately apparent that
we are only concerning ourselves with negotiation at or above the presentation layer of the OSI model. There may be quite a bit of a priori knowledge involved at the lower levels to ensure that the data are properly transmitted (i.e., packetized, CRCed, etc.) and formatted (i.e., comma separated), but this knowledge is exclusively concerned with form, not meaning or function.

Moving up the ladder of communication complexity, we arrive at content transfer. Content transfer depends on the ability to perform data transfer but adds the additional requirement of contextual agreement. That is, the sender and the receiver must have agreed upon the meaning of the transferred data such that the receiver can determine what data it has received. Furthermore, the receiver should be able to perform simple conditional logic based on the content. Whereas the recipient of a data transfer simply performs a single function upon receipt of the data, the content recipient can pass the data through a state machine and perform varied behaviors depending on what was received. Content transfer has recently become a widely discussed topic, and the extensible Markup Language (XML) was introduced largely to enable such goals. With a simple, standardized syntax in place, developers using XML are able to spend more time considering the semantics of their communications instead of their format.

Knowledge transfer represents the pinnacle of our simple classification of communications. Just as content transfer required data transfer, so knowledge transfer requires content transfer. This time, however, we expect the recipient not only to accept and process the content, but also to actually know when it has learned something new and proceed to reason about the consequences of the additional knowledge. The consequences may include such high-level notions as the mental state of the sender. The important
distinction here is that at least one of the communicating parties is capable of reasoning.

The different values for SSA-CommunicationIntent are illustrated in Table 4-1.

<table>
<thead>
<tr>
<th><strong>Value</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No data will be transferred outside of the agent.</td>
</tr>
<tr>
<td>Data-Transfer</td>
<td>Act of moving data from sender to receiver without any prior, direct or indirect, contract between the parties regarding content or meaning.</td>
</tr>
<tr>
<td>Content-Transfer</td>
<td>Act of moving data from sender to receiver such that receiver can apply simple conditional logic to discern context based on a priori contract with sender, a priori contract with third-party facilitator, and/or transferred metadata manifest.</td>
</tr>
</tbody>
</table>
| Knowledge-Transfer | Act of moving data from sender to receiver such that:  
  (1) The sender only sends the information if it wants the receiver to see it, or if it believes the receiver wishes to see it and,  
  (2) The receiver knows when it has learned something new and can proceed to reason about the consequences of the additional knowledge and why it was sent. |

4.5.1.2 Communication channel properties

A somewhat simpler characterization can be applied to an orthogonal aspect of inter-agent communication. The SSA-CommunicationIntent describes the level of contextual agreement that must exist between communicating entities. The notion of sender and receiver are introduced but are deliberately left vague. In reality, there are more definite relationships between communicating parties. The next SSA highlights these relationships. As discussed earlier in the context of granularity, agents provide our primary primitive for classification of system components. The actual implementation of the agent may include mobility, or may be realized simply as a fixed module in a constituent system. We can loosely define such agents as constructs developed by
autonomous organizations for the purpose of communicating with other agents. Given this clarification, we now present the SSA-CommunicationChannel shown in Table 4-2.

A closely related issue to SSA-CommunicationChannel is the number of expected recipients for a given message. The SSA-CommunicationChannel is used to clarify which agents act as senders or receivers. The following SSA discusses whether a message is sent to a single receiver or multiple receivers. Obviously, if an SSA-CommunicationChannel is set to none, then SSA-CommunicationMode (shown in Table 4-3) is not applicable.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Given an agent, A, and a set of agents, S, not containing A, there is no data transfer (and, therefore, no higher form of transfer) between A and any of the agents in S.</td>
</tr>
<tr>
<td>Half-Duplex Sender</td>
<td>Given an agent, A, and a set of agents, S, not containing A, all data transfer involving A originates from A and is sent to one or more of the agents in S.</td>
</tr>
<tr>
<td>Half-Duplex Receiver</td>
<td>Given an agent, A, and a set of agents, S, not containing A, all data transfer involving A originates from one or more of the agents in S.</td>
</tr>
<tr>
<td>Full-Duplex</td>
<td>Given an agent A, and a set of agents, S, not containing A, data transfer both originates from A directed to one or more of the agents in S and originates from one or more of the agents in S directed to A.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unicast</td>
<td>Given an agent A, the result of a message send is that the message, M, is sent to a single recipient agent. Also known as one-to-one communication.</td>
</tr>
<tr>
<td>Broadcast</td>
<td>Given an agent A, the result of a message send is that the message, M, is sent to all available receiving agents. Also known as one-to-many communication.</td>
</tr>
<tr>
<td>Multicast</td>
<td>Given an agent A, the result of a message send is that the message is sent to some specified subset of all available receiving agents. Also known as some-to-some communication.</td>
</tr>
</tbody>
</table>
4.5.1.3 Communication dependence

So far, we have discussed the direction and contextual agreement of communications supported by an agent. Another aspect of communications deals with the presence or absence of transactions. In general, inter-process communication (IPC), or inter-agent communication (IAC) as we may call it, can be decomposed into four primary levels. These levels provide the different values for the SSA-CommunicationDependence shown in Table 4-4.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Passing</td>
<td>An asynchronous form of communication whereby the sender transmits a message and continues without concern for receiving a response, or a receiver accepts a message without having disseminated a solicitation.</td>
</tr>
<tr>
<td>Request/Reply</td>
<td>A synchronous or asynchronous form of communication where the sender transmits a message in anticipation of a response, and a receiver accepts a message understanding that a response is requested. In the synchronous form, the sender will block until the receiver transmits a response.</td>
</tr>
<tr>
<td>Transactional</td>
<td>A synchronous or asynchronous form of communication between two or more agents where agreement must be reached towards achievement of a common goal, which, if unattained, will require possible reaction from a number of the involved agents.</td>
</tr>
<tr>
<td>Workflow</td>
<td>An asynchronous form of communication between two or more agents for the purpose of achieving a goal that may or may not require the creation of one or more transactional communications.</td>
</tr>
</tbody>
</table>

4.5.1.4 Reliability

It is important that we not confuse the SSA-Communication Dependence with the functionality provided by reliable messaging constructs, such as message queues. A message queue provides a secure, reliable location for an agent to place a message and for another agent to later obtain that message. All three levels of communication
dependence can be implemented on top of message queues, which help to provide
durability in case of processor or network failure. Other options for somewhat reduced
reliability include TCP or UDP as found in traditional network-based communication.

To assure proper consideration of the available options, we propose SSA-
CommunicationReliability (shown in Table 4-5) for describing the reliability alternatives
for communicating agents.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>A messaging agent sends a message to one or more receivers without concern for whether the message is ever received.</td>
</tr>
<tr>
<td>Network</td>
<td>A messaging agent sends a message to one or more receivers and is satisfied that the message is received as long as the network connection to the receivers is available and the receiver agents are currently operational.</td>
</tr>
<tr>
<td>Queue</td>
<td>Messaging agents send messages to receivers and require that the messages be received even in the case of temporary network failure or temporary failure of the receiving agent processes.</td>
</tr>
</tbody>
</table>

The SSA-Communication Reliability motivates the introduction of another related
SSA concerning messaging semantics. As a result of the inherent unreliability of
computer networks, it is possible that a single message may be sent multiple times or not
at all. There are a number of network protocols that address these issues, and they can
also be found in the context of remote procedure call (RPC) semantics. We introduce the
SSA-CommunicationSemantics (shown in Table 4-6) to assure that agent designers take the various options into consideration when describing a new system.

4.5.1.5 Replication

In general, replication is used to increase system availability and hopefully performance. To realize the increase in performance, it is important that the overhead required to maintain consistent replicas does not negate the benefits obtained by the proximity of local copies. It should also be noted that the choice of replication strategy has significant effects on the applicability of certain transaction models. This fact motivates us to identify the potential dependencies between SSAs. In general, an SSA should provide an orthogonal aspect of agent functionality or utilization. However, it is frequently the case that decisions made in the context of one SSA will restrict the set of possible decisions in the context of other SSAs. These dependencies are unavoidable but do not underlie any inherent problems with the model.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>All data used in the system is maintained in a single location and is shared by all agents in the system. All reads and writes by all agents in the system occur against the single copy.</td>
</tr>
<tr>
<td>Caching</td>
<td>Subsets of data used by the system are copied into caches throughout the system. If a portion of the data requested by an agent is not presently in the local cache, it is obtained by an access to the remote site that maintains the full, current version of the data. Reads and/or writes may occur against the cached copy.</td>
</tr>
<tr>
<td>Copy</td>
<td>The complete version of the data set is copied to a number of locations throughout the system. Reads and writes proceed against the local (or most local) copy.</td>
</tr>
</tbody>
</table>

The SSA-ReplicationLevel is shown in Table 4-7. In general, when replicas exist, the standard operations of read and write become more varied in their definitions. With
many, possibly inconsistent, views of the data available for access, it is extremely important that agents know what degree of timeliness and accuracy to expect from each. If there is only one copy of the data, then the problem of consistency reduces to serializing access to the data. This can be achieved through techniques such as two-phase locking (2PL), timestamp ordering, or optimistic concurrency control. If, on the other hand, there are multiple replicas of the data, then additional care must be taken to assure that agents using the replicas each see a consistent view of the data. An optimal solution would ensure that all replicas are identical at all times. However, such a solution is not possible, so a number of alternative semantics are presented.

Table 4-8. SSA-ReplicationSemantics <ACCESSTYPE P>

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Principal-Copy</td>
<td>All accesses of type P occur against a single copy maintained in a single location.</td>
</tr>
<tr>
<td>All</td>
<td>All accesses of type P must occur atomically against all replicas.</td>
</tr>
<tr>
<td>All-Available</td>
<td>All accesses of type P must occur atomically against all available replicas.</td>
</tr>
<tr>
<td>Any-Copy</td>
<td>All accesses of type P may be performed against any available replica of the data.</td>
</tr>
<tr>
<td>Quorum</td>
<td>All accesses of type P must be performed against a set (quorum) of replicas to determine the most recent data.</td>
</tr>
<tr>
<td>Gossip</td>
<td>All accesses of type P will be performed against any available replica with changes propagated in a lazy fashion.</td>
</tr>
</tbody>
</table>

An excellent analysis of the options is presented by Chow and Johnson [19], and we model our SSA-ReplicationSemantics (shown in Table 4-8) on this basis. This is our first SSA to take a parameter. In this case, the AccessType parameter is an enumeration used to overload this SSA for the generic data access operations of create, read, update, and delete.
4.5.1.6 Transactions

The requirements of a transaction are generally summarized in what are called the ACID properties. When designing a transactional system, it is important to understand these properties and to determine the appropriate level of support for each. The following outlines the four major goals of any transactional system. The relaxation of certain of these goals leads to the definition of what are frequently called extended transaction models (ETMs).

The ACID properties are:

1. Atomicity – All operations required to achieve the goal of the transaction must be completed to the satisfaction of all transaction participants, or the transaction is rolled back (undone).

2. Consistency – Regardless of the order in which the operations of the transaction are performed, the resources involved in the transaction are left in a consistent state without the violation of any integrity constraints. This does not imply that all orderings of operations are valid; instead, it requires that all valid orderings produce the same, consistent results.

3. Isolation – Concurrent transactions utilizing the same resources do not affect the consistency of one another or the integrity of the resource, even if one of the transactions fails. Furthermore, the results of activities of one transaction on a resource are invisible to any other transactions using that resource until the first transaction has committed.

4. Durability – Failure of the system after a transaction has been committed will not cause loss of the committed transaction’s work.

Before entering the realm of extended transaction models, it is important to note that not all tasks require entry into a transaction. Furthermore, different tasks may require different transactional support. The SSA-TransactionSupport, described in Table 4-9, outlines the primary options available to an agent when its services are initiated in the presence or absence of an existing transaction context.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Never</td>
<td>Regardless of the existence of a transaction, the agent receiving the request will not enroll as a member of the transaction and will not create a new transaction for itself. Furthermore, if this agent is called from a transaction the call will fail.</td>
</tr>
<tr>
<td>Not-Supported</td>
<td>Regardless of the existence of a transaction, the agent receiving the request will not enroll as a member of the transaction and will not create a new transaction for itself. If this agent is called from a transaction, that transaction should be suspended, and then restarted once the agent has completed its task.</td>
</tr>
<tr>
<td>Supports</td>
<td>When an agent receives a request that is presently part of a transaction, the agent will enroll in the existing transaction. If no transaction exists at the time the agent is called, the agent will not create a new transaction for itself.</td>
</tr>
<tr>
<td>Required</td>
<td>When an agent receives a request that is presently part of a transaction, the agent will enroll in the existing transaction. If no transaction exists at the time the agent is called, the agent will create a new transaction for itself.</td>
</tr>
<tr>
<td>Requires-New</td>
<td>When an agent receives a request, regardless of the existence of a transaction, the agent will create a new transaction for itself.</td>
</tr>
<tr>
<td>Requires-Existing</td>
<td>When an agent receives a request that is presently part of a transaction, the agent will enroll in the existing transaction. If no transaction exists at the time the agent is called, the agent will not create a new transaction, and the call will fail.</td>
</tr>
</tbody>
</table>

One approach to defining the next SSA would be to attempt a complete enumeration of all conceivable transaction models. In general, database management systems (DBMSs) use a model called serializability. However, total support of serializability is not always possible, efficient, or necessary for all application domains. To address this issue, a number of extended transaction models have been introduced over the years. In general, a new transaction model is explained in terms of how it relaxes certain of the ACID properties outlined above. This work has produced models such as partition serializability [73], relative serializability [2], partial isolation [68], relative atomicity [2], quasi-serializability [32], sagas [45], and many more. However, the names provide only minimal assistance in understanding the differences between each of the models. One
approach to solving this problem was the creation of ACTA, “a comprehensive transaction framework that facilitates the formal description of properties of extended transaction models” [21:1]. Using ACTA, it is possible to formalize the exact semantics of a particular transaction model using a set of primitive concepts. Rather than attempt to reproduce the work of the ACTA team, we base the Transaction Semantics SSA in Table 4-10 on the existence of their formalism. For each possible option added to SSA-Transaction Semantics, a complete ACTA definition for the model should be referenced.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serializability</td>
<td></td>
</tr>
<tr>
<td>Quasi-Serializability</td>
<td></td>
</tr>
<tr>
<td>etc …</td>
<td></td>
</tr>
</tbody>
</table>

4.5.1.7 Service discovery

In addition to replicating data, it is often necessary to simply move data, components, or processes from one location to another. If there are no outstanding communication sessions, this can usually be done with some form of file transfer mechanism or process migration (discussed later).

However, a problem may arise if external systems have hard-coded the location of the resource and now can no longer find it. To address this situation, naming and directory services, such as the lightweight directory access protocol (LDAP), the Microsoft Active Directory, JINI, and others have been developed. The SSA-ServiceDiscovery, shown in Table 4-11, describes the different levels of naming that agents may require.
Table 4-11. SSA-ServiceDiscovery

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>The agent is either isolated or all communications are established with statically located external resources.</td>
</tr>
<tr>
<td>Registration</td>
<td>The agent needs to utilize external resources whose locations are not known before hand, or whose locations may change during the course of processing. Also, the agent may itself move during the course of processing requiring some methodology for external processes to relocate it.</td>
</tr>
<tr>
<td>Handoff</td>
<td>The agent utilizes external resources that move while communications are in progress thus requiring in-transit messages to be forwarded from the previous location to the new location. Also, the agent may itself move during communications thus requiring some way for incoming messages to be forwarded to its new location.</td>
</tr>
</tbody>
</table>

4.5.1.8 Deployment

We have introduced many properties of multi-agent communication. Since we are dealing with software, it is important to remember the execution requirements of each of these agents. If nothing else, each agent will require a processor on which to execute its code. If several agents are communicating, then there must either be multiple processors or some preemptive capabilities in the operating system. These requirements are nothing new, but they do imply a spectrum of possible agent run-time behaviors. When designing a system, it is important to understand where each agent will be deployed and whether those deployments are static or dynamic. Allowing the processes composing the agents to be distributed across multiple processors on multiple machines, may give the system greater overall performance. Research in this area is usually done in the context of load sharing and load balancing, but it is equally important in the area of distributed objects and agent-oriented programming. The SSA-ExecutionDeployments, shown in Table 4-12, helps to distinguish a number of options for agent deployment and execution and is based on the summary in Chow and Johnson [19].
Table 4-12. SSA-Execution Deployments

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>All processing of the agent is performed on the local machine and all required services are provided by the local operating system.</td>
</tr>
<tr>
<td>Remote-Service</td>
<td>All processing of the agent is performed on the local machine, but services are requested from remote machines through use of some communication mechanism. In this way, the agent gains additional processing power from the service provider.</td>
</tr>
<tr>
<td>Remote-Execution</td>
<td>The agent can request portions of its processing to be handled by a remote host. The context of the remote executions is the same as if they were running on the agent’s local host, and there is a definable parent-child relationship between the delegating agent and its remote executions. Furthermore, the remote executions remain on the remote host until they are completed.</td>
</tr>
<tr>
<td>Process-Migration</td>
<td>The agent can move freely between accepting hosts by serializing its computation state and its communication state and moving between machines. Unlike remote execution, process migration can occur at any time during the execution of the process.</td>
</tr>
</tbody>
</table>

4.5.1.9 Persistence

It is very common for agents to utilize some persistent storage medium for maintaining their state information, configuration information, or other domain specific data. Regardless of what actions the agent or system ultimately performs with the saved data, there are a number of common options for the storage format of that data. Such options include unstructured data such as flat files, semi-structured data such as the extensible Markup Language (XML), or structured data such as information stored in a relational database schema or in an XML file with an explicit XML-Schema. The SSA-PersistenceFormat, shown in Table 4-13, will introduce these options but will not attempt to describe the nuances or variability inherent in each of the choices. Further detail on storage formats can be found in a wide array of textbooks.
Table 4-13. SSA-PersistenceFormat

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unstructured</td>
<td>A proprietary storage format lacking a declarative metadata description that could be shared among autonomously developed agents.</td>
</tr>
<tr>
<td>Semistructured</td>
<td>An organized data format that has at least an incomplete metadata description for distribution among autonomously developed agents. In this context, the term incomplete refers to the fact that at least a portion of the persisted data is accessible through automated processing based on the metadata description. If there are additional data, they do not negatively affect the availability of the data recognized through use of the metadata. Conversely, a metadata description of an existing data format does not guarantee that the data or surrounding infrastructure for the data exists, and agents are expected to handle any errors occurring as a result of missing information.</td>
</tr>
<tr>
<td>Structured</td>
<td>A highly organized and reasonably static storage format with a precise and complete metadata description of the available data. The format of all data available to an agent is described in the metadata, and if a data format is described in the metadata then an agent can be guaranteed that at least the surrounding infrastructure for the data is in place (of course, there may be no records).</td>
</tr>
</tbody>
</table>

4.5.1.10 Presentation

Up to this point, we have discussed a number of generic topics related to answering the “what” facet of a 6FR. Before we tackle another facet, we introduce one last SSA regarding the presentation used by the agent.

Table 4-14. SSA-Presentation

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>The agent exposes a programmatic interface that can be used by developers or other agents to exercise the functionality provided by the agent.</td>
</tr>
<tr>
<td>Command-Line</td>
<td>The agent exposes a command line interface for script based or user interaction.</td>
</tr>
<tr>
<td>Desktop</td>
<td>The agent exposes a desktop graphical user interface for user interaction.</td>
</tr>
<tr>
<td>HTTP</td>
<td>– The agent exposes an interface that can be accessed through a standard WWW browser.</td>
</tr>
</tbody>
</table>
SSA-Presentation, just as with all of previously presented SSAs, different agents within a constituent system often select different options for the same SSA. In the case of presentation, it is common for an agent to expose both an API and a user interface. The SSA-Presentation, shown in Table 4-14, outlines these and other presentation options.

4.5.2 Facet “Where”

Proceeding on through our list of facets, we next examine where an agent will be deployed. There are a number of ways to answer this question. One solution entails discussion of which enterprises will ultimately utilize this agent. This is a valid solution, and an important question, but we answer it in the context of “who” rather than “where.” Instead, we choose to consider the connectivity attribute of where the agent will be deployed. This is shown in the SSA-Connectivity illustrated in Table 4-15.

Table 4-15. SSA-Connectivity

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isolated</td>
<td>This agent is installed on a standard personal computer without access to the Internet.</td>
</tr>
<tr>
<td>Connected</td>
<td>This agent is installed on consumer-outfitted desktop computers with at least dial-up access to the Internet.</td>
</tr>
<tr>
<td>Workstation</td>
<td>This agent is installed on business-outfitted desktop workstations with constant LAN access to the Internet.</td>
</tr>
<tr>
<td>Server</td>
<td>This agent is installed on business-outfitted servers with constant LAN access to the Internet.</td>
</tr>
<tr>
<td>Laptop</td>
<td>This agent is installed on laptops with long periods of both connected and disconnected operation.</td>
</tr>
<tr>
<td>PDA</td>
<td>This agent is installed on personal digital assistants (PDAs) with intermittent network connectivity (usually through wireless access).</td>
</tr>
</tbody>
</table>

4.5.3 Facet “Who”

Looking at the SSA-Connectivity, there is no doubt that the options will have tremendous impact on the design of the agent. A company would be foolish to design an
agent for a PDA using the same techniques employed when designing for the server market. In the server domain, there is no need to consider issues, such as hand-off between service providers resulting from user mobility, but there is a need to utilize all available processors present in the server. Just as important as where the agent is deployed, is to whom it is deployed. The SSA-UserTypes, shown in Table 4-16, attempts to categorize the different types of users a system may strive to accommodate.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private-User</td>
<td>A PrivateUser is employed by the same organization as the team developing the agent, or has contracted with a development team for a long-term exclusive contract. The PrivateUser is the primary, or possibly only reason for development of the agent.</td>
</tr>
<tr>
<td>Corporate-User</td>
<td>A CorporateUser is employed by a separate organization from the team developing the agent but has signed a long-term service contract with the development team including both maintenance and addition of new features.</td>
</tr>
<tr>
<td>Public-User</td>
<td>A PublicUser maintains no affiliation with the team developing the agent and likely will have no contact with the team.</td>
</tr>
<tr>
<td>Corporate-Developer</td>
<td>A CorporateDeveloper is employed by a separate organization from the original agent development team. CorporateDevelopers differ from the CorporateUsers in that the former not only use the agent but may also extend the agent through their own efforts. Some level of interaction is usually maintained between the CorporateDevelopers and the original agent developers, and the CorporateDevelopers may help to drive the evolution of the agent.</td>
</tr>
<tr>
<td>Public-Developer</td>
<td>A PublicDeveloper develops or personalizes constituent systems based on the original agent but has minimal access to the original developers. Public developers expect a stable agent and assume that other unaffiliated PublicDevelopers also have access to the same agent.</td>
</tr>
</tbody>
</table>

The question of whom a particular agent is meant to support can help to determine the placement and presentation of that agent in the overall system offering. For instance, if PublicDevelopers are the intended audience, then the agent will usually expose a stable
well-known API. On the other hand, if the agent is meant only for PrivateUsers, then the stability of the API (if there is one) is not as important as the timely delivery of all the necessary functionality required to support the everyday business operations.

### 4.5.4 Facet “When”

The “when” facet of the 6FR is intended to highlight the estimated maintenance efforts for a particular agent.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed</td>
<td>The requirements of this agent never change. Requirements changes cause redesign, re-implementation, and redeployment. That is, new requirements imply replacement of the old agent, not upgrade.</td>
</tr>
<tr>
<td>Scheduled</td>
<td>The requirements of this agent are expected to change at regular intervals during the lifetime of the product. An example would be the tax tables used in a tax calculating application. Requirements change cause re-implementation and redeployment. That is, new requirements place developers back in the development environment to upgrade the existing agent.</td>
</tr>
<tr>
<td>Unknown</td>
<td>The requirements of this agent are not considered stable and are likely to change frequently. There is no formal understanding of the requirements, only a vague need outlined in the definition of the constituent system. Requirements change causes self-adaptation or requires the inherent ability to accommodate the specific alterations.</td>
</tr>
</tbody>
</table>

To support the need for this analysis, it is useful to draw an analogy between the design of an automobile and that of a software system. Consider the location for the oil dipstick in a car. If the manufacturer placed the dipstick under the engine, it would be very difficult to check the oil level. Since checking the oil is a common task that must be performed many times a year, car manufacturers choose to make the dipstick as accessible as possible. The point we are making is that understanding the expected maintenance schedule for the various aspects of a software system allows the designers of
the system to ensure that the most commonly altered portions are the most loosely coupled. In the software world, loose coupling of modules is akin to easy physical accessibility of automobile parts. The SSA-MaintenanceSchedule, shown in Table 4-17, provides a basis for answering these questions.

4.6 Requirements Patterns

At this point, it should be very clear that we have located ourselves in the context of distributed systems. Rather than trying to reinvent the standard approach to requirements documentation used for centralized system analysis, we have attempted to abstract the issues that differentiate both centralized and distributed development efforts. In the remainder of this section, we show the relationship between SSAs and requirements patterns (RPs). In particular, we show how selections over a set of SSAs are used to create a generic picture of high-level component (agent) requirements or, as we call it, an RP.

4.6.1 Example RP for a Centralized System

Taking selections from a number of the SSAs in combination leaves us with a high-level picture of a system’s architecture. To provide a baseline, we first look at a pattern used in many standard centralized desktop applications.

RP-IsolatedDesktopApplicationWithTransactions

1. SSA-CommunicationIntent(None)
2. SSA-CommunicationChannel(None)
3. SSA-TransactionSupport(Requires-New)
4. SSA-TransactionSemantics(Serializability)
5. SSA-ReplicationLevel(None)
6. SSA-ServiceDiscovery(None)
Looking at the pattern, RP-IsolatedDesktopApplicationWithTransactions we see that a number of available SSAs have been disregarded, and a number of those included have defaulted to the least demanding values. A good example of such a system is a single-user database application utilizing a proprietary database format presents the user with a graphical interface. More specifically, an instance of this application may be an off-the-shelf pest-control customer and inventory database. The transactions are isolated to the application itself but are needed to ensure that purchase of additional inventory is accompanied by debit of the allocated inventory finances. Many other examples of this type of application could be given, but the important point to realize is that we have identified a requirements pattern (RP). Once we have established a name for this RP, we can begin to associate existing systems with the pattern, and furthermore, we can begin to identify this pattern with new development efforts.

4.6.2 Application of Requirements Patterns

While not the topic of our immediate research, it is important to describe the implications of identifying and using requirement patterns. Consider a database containing the project metadata associated with developing systems conforming to specific RPs. Project metadata includes such elements as the length of development time for the various components of the system, what the components were, what level of
expertise the developers had when they began, and what tools and programming languages were used. If this project metadata is maintained in conjunction with the RPs of the developed system, then future development efforts can take advantage of this data to better predict product feasibility, cost, and time to completion. In simplified form, the product manager would input the RP describing the agent he intends to build, and the project database would output metadata describing what he should expect in terms of overall development effort, budget, and other managerial details. We propose that one of the most fundamental reasons for resistance to structured software engineering techniques is that management does not see an immediate return. Perhaps, the existence of such a system would help to reverse this.

4.6.3 Stretchable Requirements Patterns

As stated at the start of this dissertation, and reiterated in our introduction to SSAs, the motivation for developing these new foundations is to provide a firm basis for the description of stretchability and to ultimately formalize the notion to the greatest extent possible without resorting to esoteric mathematical notations. One way of doing this is to define stretchability in terms of a set of SSAs. As shown in the discussion of the RP-IsolatedDesktopApplicationWithTransactions, a set of SSAs instantiated with chosen values leads to the creation of a requirements pattern (RP). Our goal, then, is to identify sets of SSAs that imply a stretchable system. Such sets compose what we call stretchable requirements patterns (SRPs).

4.6.3.1 One more “When” facet SSA

Stretchability helps to simplify and support cross-enterprise reuse, maintenance, and communication. Once again we see the appearance of the term cross-enterprise underscoring the relation to distributed systems. In essence, stretchability defines the
characteristics of constituent systems allowing composing agents to evolve independently with minimal impact on the global system. In fact, the definition of a (global) system shifts from a large, yet bounded, set of modules to a more indeterminate structure whose size and composition varies constantly with time. The number and types of changes that can be supported help to define the spectrum of stretchable solutions. The SSA-SupportedModifications, shown in Table 4-18, outlines the types of changes that a system may be forced to accommodate. This issue becomes important for SSA-MaintenanceSchedule(Scheduled) and SSA-MaintenanceSchedule(Unknown). However, in the context of stretchability, it is SSA-MaintenanceSchedule(Unknown) that is of particular interest. That is, how can software be developed to enable runtime support of the following list of changes? Please note, that this list will likely be extended, and the ultimate goal is to support all eventual extensions.

All of this adaptability to change becomes a necessary requirement when we begin to accept the view of the global system as a constantly evolving set of autonomously developed agents. There are essentially two conflicting requirements. First, each enterprise wants its constituent systems to be able to communicate with other enterprises’ systems without having to perform large-scale, or even minimal, changes to its existing processes. Second, and very much at odds with the first requirement, each enterprise wants to develop its constituent systems in isolation without being constrained by the choices or changes occurring elsewhere.
Table 4-18. SSA-SupportedModifications

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InterfaceName</td>
<td>Changes to the names of the interfaces exposed by an agent.</td>
</tr>
<tr>
<td>InterfaceNumber</td>
<td>Changes to the number of interfaces exposed by an agent.</td>
</tr>
<tr>
<td>MethodSignature</td>
<td>Changes to the number and types of arguments passed to a method (passed in a message).</td>
</tr>
<tr>
<td>MethodImplementation</td>
<td>Changes to the implementation of the methods.</td>
</tr>
<tr>
<td>TransactionParticipantNumber</td>
<td>Changes to the number of participants involved in a transaction.</td>
</tr>
<tr>
<td>TransactionParticipantType</td>
<td>Changes to the types of participants involved in a transaction.</td>
</tr>
<tr>
<td>TransactionParticipantLocation</td>
<td>Changes to the locations of the parties involved in the transactions.</td>
</tr>
<tr>
<td>DataResourceLocation</td>
<td>Changes to the location(s) of data (replicas) involved in the transaction.</td>
</tr>
<tr>
<td>ServiceResourceLocation</td>
<td>Changes to the locations of the services enlisted by the agent.</td>
</tr>
<tr>
<td>ConnectionStatus</td>
<td>Changes to the connected status or availability of the parties involved in the transaction.</td>
</tr>
<tr>
<td>PersistanceFormat</td>
<td>Changes to the physical storage structure used for data persistence and transaction state.</td>
</tr>
<tr>
<td>Language</td>
<td>Changes to the language in which the agents are written.</td>
</tr>
<tr>
<td>Platform</td>
<td>Changes to the underlying architecture on which the agents are developed.</td>
</tr>
<tr>
<td>Security</td>
<td>Changes to the security settings of data and services involved in a transaction.</td>
</tr>
<tr>
<td>etc…</td>
<td></td>
</tr>
<tr>
<td>All</td>
<td></td>
</tr>
</tbody>
</table>

4.6.3.2 Approaching a stretchable requirements pattern

The purpose of this section is to visit each of the SSAs presented above and to examine the proper instantiations in the context of a stretchable system. A stretchable system is meant to support cross-enterprise interoperability, as well as runtime adaptation to a variety of possible changes. As we progress through each of the SSAs, it should become clear how the explicit representation of each concept helps to clarify the intended functionality of the final system. Ultimately, we will be left with a stretchable...
requirements pattern (SRP) capable of serving as the basis for constituent system design and as motivation for requirements of a global framework.

**Communication Intent.** The primary type of information transferred between enterprises, or between an enterprise and a customer, is domain specific data related to completing a sale or providing a service. The customer will likely want to know pricing information, scheduling information, or general product data, while the enterprise will need to obtain customer information such as credit card numbers, current insurance coverage, and other personal details. When one considers that services can range from buying a car, to purchasing automobile insurance, to seeking medical assistance, it is obvious that the domain data will vary greatly. Furthermore, not every company or service provider will want to represent its data in the same fashion. Given these facts, we choose to instantiate the SSA-Communication Intent with the value, Content-Transfer. This choice implies that some a priori communication is needed between the communicating entities, or at least between each communicating entity and a third party facilitator. In either case, the goal is to somehow establish a common ontology to provide semantic context for the intended conversations. One possible approach is the use of a third-party facilitator. If the third party maintains a well-known specification, then all other parties need only maintain a mapping between their proprietary representations and the well-known version.

It should be noted that for the purposes of stretchability, we have chosen to relax the need for Knowledge-Transfer. While interpretation of a receiver’s mental state may prove imperative for certain applications, it is not a strict requirement for useful and profitable communication.
**Communication Channel Properties.** There is little doubt that the agents in a constituent system require SSA-Communication Channel (Full-Duplex). The need for broadcast or multicast is not quite as immediate. It is clear that unicast communication can be used to duplicate the functionality of either broadcast or multicast. However, this use of unicast leads to inefficient network bandwidth utilization as well as slower communications. For these reasons, the choice to use improved algorithms for the communication mode is left to the constituent system designers, and SSA-Communication Mode (Unicast) is selected as a sufficient option for stretchable systems.

**Communication Dependence.** Within a global system, agents are used to accomplish many different types of functionality. In general, the services of many independent agents are enlisted to realize the goals of a given task. Some of these goals, such as a monetary transfer, require the strict semantics of serializable transactions. Other goals only require continuous propagation of information from one agent to another with some control information to ensure progress is always made toward achievement of the goals. In terms of our SSAs, a typical stretchable system must support SSA-Communication Dependence (Workflow). This choice mandates the need to support transactions, but also implies that transactional communications are required as building blocks of the more complicated high-level tasks.

**Reliability.** In terms of messaging, it is probably possible to lower the reliability requirements and still perform useful work. However, recent literature seems to point to exactly-once, durable messaging as the standard for electronic business transactions. Given this fact, we choose SSA-CommunicationReliability(Queue) and SSA-
CommunicationSemantics(Exactly-Once) with the caveat that certain agents may successfully relax these requirements.

**Replication.** For performance reasons, and sometimes to avoid the need for mobility, data may be replicated at various sites. A stretchable form of replication would allow the structure of the replicated data to differ at each location. The goal is to recognize that all replicas identify the same physical (or logical) concept without forcing each replica to store all of the same information regarding that concept. For example, an insurance site stores a person’s insurance information; a medical site stores that person’s medical information; both sites should be able to reconcile that the person in question is a single entity. This avoids the problem of object schizophrenia that currently plagues object-oriented data representations. The need for replication is inherent in cross-enterprise applications, and so we choose SSA-ReplicationLevel(Copy). This does not directly address the issue of object schizophrenia, but it does highlight the requirement for maintaining consistent copies of the same data across enterprise boundaries. It should be noted that SSA-ReplicationLevel(Cache) may be applied to increase performance and to enable disconnected operation in certain situations.

It should be clear that in a distributed environment, the probability that all replicas will always be available is unlikely. For these reasons we can dismiss SSA-ReplicationSemantics<P>(All) for P in \{create, read, update, delete\}. For performance reasons, quorum may not always be the optimal solution, but it may be necessary if gossip does not meet the application’s consistency requirements. Therefore, we leave this open and select SSA-ReplicationSemantics<Read>(Any-Copy, Quorum) and SSA-Replication<P>(Quorum, Gossip) for P in \{create, update, delete\} as the most suitable
candidates. Certainly other options are useful, and choosing different options provides variable versions of stretchable requirement patterns and thus different implementations of constituent systems.

Transactions. If we review the list of changes a stretchable system is supposed to support, we see that many of those changes are related to transactions. In the domain of cross-enterprise, distributed systems it is very important that communicating entities all agree to commit to the deal they have negotiated. It would be unacceptable for a supplier to ship product without ensuring that payment was satisfactorily completed. Following this line of reasoning, it is logical to assume some degree of support for each of the ACID properties. In a distributed system, failures have a far greater probability of occurring, so durability is absolutely essential. Furthermore, with the potential for hundreds or thousands of transactions occurring at any given time, it is important to maintain the properties of isolation and consistency (serializability). It should be noted, however, that in certain circumstances, these semantics could be relaxed without compromising the utility of the system. Furthermore, other circumstances may force relaxation of these requirements, for instance, disconnected operation. A particularly good and stretchable solution is to abstract the transaction model from the agents and allow any agent to work within the confines of any transaction model. However, this may not always be possible. In general, an agent should declaratively specify its transactional requirements. Thus we choose no particular value for SSA-TransactionSupport since this will vary for each agent. Furthermore, the choice of using the standard transaction model, or some extended model should be left to the agent developers. However, a stretchable
framework should support any transaction model, and an agent should strive to support multiple values for SSA-Transaction Semantics.

**Service Discovery.** Based on the SSA-SupportedModifications, the location of data and processes should be subject to change without adversely affecting the global system (or the constituent systems within which it is contained). The primary means of achieving this goal is through the use of a naming service that maintains a unique identifier for the resource along with its present location. For the majority of applications, SSA-ServiceDiscovery(Registration) will likely be sufficient; however, in the case of mobile agents, or wireless devices used during travel, SSA-ServiceDiscovery(Handoff) will be required.

**Deployment.** A stretchable system should essentially be able to handle all of the options. Thus we make no particular choice for the SSA-ExecutionDeployments. While running on a personal digital assistant (PDA), an agent of a constituent system should be able to perform useful work even in a completely disconnected state. When the connection is reestablished, the agent should be able to determine its new location and retrieve any messages that may have been sent to it during the disconnected period. Furthermore, while connected, the agent should be able to utilize the resources available on any other processors in the system. The use of mobile agent technologies also implies the need for process-migration.

**Persistence.** The SSA-Persistence Format highlights the primary options available for writing data to durable storage. The main distinction of stretchable systems is that the persisted data is subject to constant changes in the location where it is stored, what is stored, and how it is stored. As the information in the particular domain evolves, so must
the representation format used for maintaining the new data. In situations such as this, the static nature of relational database schemas often proves unwieldy. The first industrial attempts to remedy this problem took the form of object-relational databases (e.g., Oracle 8i) and pure object-oriented databases (e.g., POET). Another alternative that has recently generated great interest is semi-structured data storage (e.g., Tamino). If Stanford University’s Lore project could be used as an indicator, we could conclude that ultimately the lessons learned in developing the object-oriented databases yield their greatest importance when applied to improving the design of semi-structured databases. With the ever-increasing ubiquity and obvious flexibility of XML, we feel that stretchable systems will be based on SSA-Persistence Format (Semistructured), but that SSA-Persistence Format (Structured) will remain extremely important as the primary means for storing and querying large amounts of relatively static data (static in terms of structure, not content).

**Presentation.** The interfaces provided by a constituent system will be many and varied. Once again, we must consider the many different scenarios under which the system’s agents will be used. When graphical mobile agents traverse the Internet to greet a user on her PDA, they will be expected to provide an interface amenable to the constraints of the PDA’s display. This interface may be a standard GUI or it may be constrained to the limited output available for texting applications. Other agents of the system may not provide any user interface at all, but instead may serve to enable system extension or utilization through programming or scripting languages. One of the ways we can isolate which different agents in a constituent system require their own requirements pattern is to analyze the different interfaces that the agents provide. This form of system partitioning
may help to isolate dependencies that should be removed to keep the agents as decoupled as possible. If agent migration ultimately becomes standard practice, then it will also be useful for agents to be able to select which interface they wish to take with them, or at the very least, for each agent to have an interface. Considering the many possible scenarios, it is clear that each agent will support one or more of the options in SSA-Presentation dependent on their ultimate utilization.

**Connectivity.** The same line of reasoning applied to SSA-Presentation can be used to establish selections for SSA-Connectivity. Different agents of each constituent system will be subjected to different connectivity requirements. It is important to distinguish the particular circumstances under which each agent will operate and assign the appropriate selections on that basis. Thus, a stretchable system will have support for all of the options presented in SSA-Connectivity, and a stretchable agent will strive to support as many options as possible.

**User Types.** In the case of SSA-UserTypes, we once again see a situation where the different agents in a constituent system will have different requirements; however, we can state with certainty that the global framework must support SSA-UserTypes(Public-Developer). This is reasonable since the primary feature of a global framework is autonomous extensibility. To make the global framework accessible to the greatest number of people, it should also support User Types Public-User and Corporate-User. The remaining two options of SSA-UserTypes are not directly applicable because there is no primary developing enterprise of a global system. There are, of course, the original architects of the global framework, but they have no direct involvement with any of the domains or constituent systems that eventually emerge within the global system. It
should now be clear that constituent systems are developed by extending the functionality provided by the global framework.

**Maintenance Schedule and Supported Modifications.** The final two SSAs discuss maintenance schedule. It should be evident that from the perspective of the global system, the global framework must support SSA-Maintenance Schedule (Unknown). The fact is that, only after the original global framework is fully implemented and deployed, does the creation of the global system actually begin. Furthermore, the eventual information contained in the system is completely unspecified when the framework is being developed.

The agents of a constituent system must support SSA-Maintenance Schedule (Unknown). One of the primary goals of designing stretchable systems is enabling the runtime adaptation of agents against a multitude of possible modifications. The real key to stretchability is a high level of support for SSA-Supported Modifications. Since the modifications outlined in that SSA imply the possibility of transactions, replication, service discovery, and many other features, it is also imperative that agents support these concepts. The goal of this dissertation is to explicitly detail the many features required for development of agents within a constituent system and to provide an unambiguous means of representing the requirements of such agents. The culmination of the effort appears in the next section where we introduce a first stretchable requirements pattern.

**4.6.4 Stretchable Requirements Pattern**

Throughout this section, we have discussed the requirements of stretchable systems in terms of the SSA options they need to support. We now present one possible list of selections and submit it as the requirement pattern of a highly stretchable mobile agent. Using only the SSAs presented, we submit that many other stretchable requirements
patterns (SRPs) are possible. The following SRP simply provides one interesting example.

RP-HighlyStretchableMobileAgent

1. SSA-Communication Intent (Content-Transfer)
2. SSA-Communication Channel (Full-Duplex)
3. SSA-Communication Mode (Unicast)
4. SSA-Communication Dependence (Workflow)
5. SSA-Communication Reliability (Queue)
6. SSA-Communication Semantics (Exactly-Once)
7. SSA-Replication Level (Copy)
8. SSA-Replication Semantics <Create> (PrimaryCopy)
9. SSA-Replication Semantics <Read> (PrimaryCopy)
10. SSA-Replication Semantics <Update> (PrimaryCopy)
11. SSA-Replication Semantics <Delete> (PrimaryCopy)
12. SSA-Transaction Support (Requires)
13. SSA-Transaction Semantics (ACTAFormalized(Serializable))
14. SSA-Service Discovery (Handoff)
15. SSA-Execution Deployments (Process-Migration)
16. SSA-Persistence Format (Semistructured)
17. SSA-Presentation (API)
18. SSA-Connectivity (PDA)
19. SSA-User Types (Public-Developer, Public-User)
20. SSA-Maintenance Schedule (Unknown)
21. SSA-Supported Modifications (All)
4.7 Summary

This chapter started with the express goal of elucidating the concept of stretchability. Since stretchability is not a quantitative measure, its definition had to be achieved without the assistance of mathematical rigor. This requirement has both pros and cons. Mathematical formalism helps to avoid ambiguity and affords its user a rich body of existing work on which to base new ideas. Furthermore, establishing a formal representation for an idea allows any associated assertions to be proved. There is no doubt that certain aspects of stretchability will eventually benefit from a more formal treatment, but presently it is more important that the idea be clearly presented and made accessible to the greatest number of people. For that reason, we have taken a middle ground and sought to express the primary tenets of stretchability in as unambiguous a fashion as possible without burdening the reader with esoteric notation. The introduction of the six faceted requirement (6FR), software specification axioms (SSA), and requirements patterns (RP) has helped realize this intention.

Stretchability covers a wide scope of computer science topics. One of the most difficult aspects of defining stretchability is bringing together this abundance of disparate information and illustrating the common thread that runs throughout. The source of this challenge may stem from the fact that stretchability is as much a characteristic of software systems as it is of software itself. Furthermore, the systems that stretchability seeks to characterize exist on a global scale. To help clarify this requirement, and to provide a firmer foundation for the remainder of the chapter (and dissertation), the concepts of global system and constituent system were introduced. It is important to understand the relevance of the global system and why its definition is imperative when discussing stretchability. Stretchable systems are not developed to run in isolation. In
fact, without the supporting framework of a global system there simply is no
stretchability. The realization of the global framework, whether as a standards document
or as a set of executables and libraries, is left open and constitutes one area of future
research.
CHAPTER 5
CONTENT-BIASED COMMUNICATION

As humans, we are intimately familiar with the use of communication. In fact, people are so inherently drawn to communicate that one may imagine that the act provides some form of sustenance. For a social species such as ours, the importance of self-expression and information-transfer cannot be understated, and perhaps may help to explain our rather voluble nature, but it may also help to explain the tremendous complexity of natural languages. If humans had simply devised language as a tool for communicating informative details pertinent to a given situation, then the resultant grammar may have been surprisingly more succinct, if not bland and expressionless. However, we did not choose this route, and subsequently we are gifted with literature, philosophy, and music – as well as the questionably pleasing sounds of everyday conversation.

5.1 Do Unto Others Only What They Ask

Of course, one may wonder how all of this is relevant to the issue at hand. Based on the work of the previous chapters, it should be clear that the stretchable systems we seek will likely be based on software agents. It is at this point, where we must decide the level of anthropomorphic behavior we shall bestow upon those agents. If we follow convention, we might be tempted to describe a system where each agent is an autonomous entity with a mental state sustaining such ideas as beliefs, desires, and intentions. While this is a path of admirable intent, it may not ultimately lead to the most pleasing solutions.
I submit, that if we engender our agents with mental states, then we endanger our prospects for honest, purposeful communications. I further suggest that the majority of human language is dedicated to the purposes of evasion and dishonesty, with secondary importance placed on comforting our kinsmen. The additional purposes, including the transfer of relevant information, rank far lower when measuring the motivations for language design. One clear example in defense of this hypothesis can be seen in Brown and Levinson’s politeness principles and the concept of face.

If we take Searle’s speech act, defined in Chapter 2, as a satisfactory representation of the dimensions of an utterance, then elimination of an agent’s mental state can be equated to trivializing the perlocutionary act. For example, if a man asks his wife *What color is that dress?*, she will not reply with *Why, do I look fat?*. That is, rather than attempting to uncover her husband’s mental state (his intention for asking), the woman will simply respond to the illocutionary act directly questioning the color of the dress she is presently wearing. It is important to note that the remaining dimensions of the speech act are still all important. In fact, to ensure we have a reference act, and not an act of predication, it is imperative that “that dress” actually refers to something equivalent from the perspective of both participants.

Eliminating any reasoning regarding the mental state of the agents in a system has greater effect than simply negating the importance of the perlocutionary act. Because we no longer have to concern ourselves with honesty, non-conventional effect, and politeness, we are free to examine a more primitive form of communication used exclusively for information transfer. We can utilize constructs, such as a communal mind for simplifying the maintenance of context throughout a conversation. Furthermore,
since agents no longer attempt to ascertain mental states of other participants, we can eliminate the TELL performative and minimize the importance of felicity conditions. Agents now operate under the philosophy “do unto others only what they ask.” Thus, agents do not volunteer information that has not been requested. To do so would imply that the sender thinks the receiver may find that information interesting, but the sender does not think about the receiver’s mental state, so such a situation is contradictory.

5.2 Language for Content-Biased Communication

Even given the simplification obtained through elimination of mental state, the problem of initiating and maintaining progressive conversations that ultimately lead to some useful purpose is still quite challenging. The first major difficulty arises in enabling all participants to agree on the meanings of all lexemes in the conversation. For instance, when I speak of my “car,” it would be impossible for me to communicate with someone who pictured a “dog” in his mind. The example may seem absurd, but it is important to realize that the assignment of semantic meaning to the word “car” is completely arbitrary, and only successful as a result of social convention.

A second difficulty surfaces when attempting to identify the referent of any diexes in an utterance. When I refer to “that car,” it is imperative that all participants in the conversation recognize exactly which automobile I am describing. Another, well-known obstacle, is formalizing a scaffolding mechanism upon which one may construct dialog concerning new aspects of reality. That is, even if we choose to model the universe, not in its entirety, but only in slices as physical-behavioral units, we are left with the questions of how we represent the seemingly infinite number of objects, relationships between objects, granularities of objects, and contexts that arise in everyday situations? Furthermore, how can time (and evolution) be factored into the model? These problems
comprise the goals we seek to define in a new content-biased language, and are summarized in the following list:

1. How can we obtain agreement on the semantic meaning of lexemes?
2. How can we establish the unique identity of a referent?
3. How can we represent any concept at any level of granularity?
4. How can we represent any relationships that may occur between concepts, including relational, non-relational, Cambridge, and comparative relations?
5. How can we distinguish the use of the same concept in different contexts without always discussing all aspects of the concept and without losing the importance of the specific context?
6. How can we factor time and object evolution into the model?
7. How can we construct the scaffolding such that any model of any physical-behavioral unit can be modified and reused, including direct reference from any other model of any other physical-behavioral unit?
8. How can we enable a means of expressing the rationale or intentions behind an individual’s decision to make data accessible?
9. How can we ensure that the description of intentions with content is orthogonal to the content itself?
10. How can we ensure that partial content is understood as a subset of total content, and the functionality of an agent is not inexorably halted when presented with partial content?

In the remainder of this chapter, we examine a number of alternative constructions for a content-biased language suitable for use as an Interlingua between autonomous, semi-cooperative, non-mentalistic agents in a global system. Throughout the process, many desirable language properties are identified and help constrain the search. In the end, one possible construction will be proposed. While this final model may not solve all conceivable problems, it clearly addresses numerous issues, and serves well as a basis for continued investigation.
5.3 Language as a Type of Metadata

At this point, we have established the need for a new language capable of describing models of physical-behavioral units for any domain of discourse. In addition, we have outlined the foundational assumptions for the construction of the language. In general terms, this language will be used to convey information between agents. As such, it is important to understand the relationship between the language and the agents that will use it. It is important to understand that the language is a carrier for data, as much as it is a semantic mapping between slices of reality and purposefully defined lexical units. In the realm of computers, data may be roughly categorized by how it is manipulated. Some data is stored in large, structured repositories that are analyzed to find trends or particular points of interest. Other data is mapped within the address space of a program to represent the state of the process and the information upon which it is calculating. Finally, data is also used as a communication mechanism between processes (or functions) to provide particular information to algorithms calculating a useful result or performing some other needed action.

In general, these three uses of data are separated by a distinction in their metadata descriptions. That is, when data is read from a database into the address space of a program, it is converted from the relational-based schema at the source, to the type-system-based schema of the destination. Likewise, when data is transferred among the processes in a distributed system, it undergoes a conversion from the source schema, through an intermediary, and finally to the destination schema. To realize stretchable agents, it is useful to retain the former distinction with qualification and to completely remove the latter distinction. That is, the relational (or otherwise) schemas of databases shall have their atomic units mapped to the lexemes of the language, while the type-
systems used by the agents will be the language. Thus, the language we are proposing runs a fine line between an ontology and a type-system. It is an ontology that models aspects of the world but provides no support for reasoning. It is a type-system that is used directly by programming languages, but it is external to the languages that use it. It is important to realize that information may be stored using the language, but in most cases, especially for large aggregates of data, this will not be the optimum practice. This makes sense when one considers that telephone books are structured as flat files, instead of using natural language. The following set of definitions helps clarify the distinction between the metadata types described above:

1. Constraint Metadata – places constraints on the values that may appear in instances based on the metadata description,
2. State Metadata – structure used to maintain information related to the state of a process,
3. Storage Metadata – describes the storage structure and locations of a set of related data items, and
4. Communication Metadata – maps a globally accepted semantic interpretation to a uniquely definable position in an abstract structure.

**5.4 External Open Ontological Type System**

In general, a language is composed of two distinct components, the vocabulary and the grammar. The vocabulary defines the lexical elements of the language, and the grammar defines the syntactical rules for combining the elements. The semantics implied by certain syntactically valid constructions form yet another important dimension of language. In the case of our content-based language, the vocabulary will be referred to as an External Open Ontological Type System (EOOTS). This follows directly from the fact that this language serves as a type system but also has properties of an ontology.
When an agent receives a request-for-content, it must understand what information the sender is requesting. Likewise, when that agent returns a reply-with-content, the original sender must understand the response. There are a number of approaches that can be applied to achieve this goal. The immediate solution is to create a standardized content representation vocabulary that all communicating agents must use. Note that a markup language such as XML is not, in and of itself, sufficient to achieve this goal. Just because an agent can utilize a standard XML parser to extract the data from the XML document, does not mean that the data points have any particular significance. Thus, the problem we are trying to solve is not one of simply parsing the individual data values within a message, but rather one of comprehending the significance of those values. That significance is conveyed by the semantic layer logically situated atop the EOOTS.

5.5 Comparing Content-Biased and Service-Biased Communication

Using the process of human conversation as a paradigm, one can imagine a system in which agents communicate primarily through the transfer of content. Each agent performs its tasks in relative autonomy and communicates with other agents primarily when additional content is required for continuation of the individual’s process. We submit this in stark contrast to the more service-biased approach suggested by the evolution of RPC mechanisms (e.g., SOAP, WSDL, UDDI). It is our opinion that both approaches are worthwhile but that the content-biased approach may prove more applicable in many circumstances. The following definitions of Inter-Agent Communication Paradigms help to clarify the distinction between the two models of communication:

1. Service-Biased Communication – given some consumer agent, A1, a communication from A1 to a provider agent, A2, entails a request for service by means of a fixed-signature method call on a well-known interface exposed by A2.
This service may perform a task as simple as returning a single data value, or as complex as spawning a workflow.

2. **Content-Biased Communication** – given some participant agent A1, a communication from A1 to another participant agent, A2, entails a request for content by means of a free-form message send to a well-known interface exposed by A2. Acquisition of the requested content by A2 may simply require accessing a single data value, or may entail initiation of a complex workflow.

In the service-biased case, it is assumed that the provider agent will never require additional information from the consumer agent. As such, there is no standard mechanism for a callback to the consumer. In fact, the very use of the term callback shows the pervasiveness of the service-biased approach. The reality is that we want more than a callback; we want the possibility of an open dialog between the communicating agents. Whereas in the service-biased approach, there must be a fixed-signature method in the interface to implement each desired functionality, in the content-biased approach the interfaces are open and reflect a more anthropomorphized communication. As each agent executes its workload, it is free to query any other agents about content that may be useful. This means that all content required to perform a given task need not be supplied up-front. Rather, the content is requested when needed. This reveals the possibility of pausing and serializing the agent communication while an ancillary GUI agent (discussed in the following chapter) is dispatched to obtain additional content from a human user. Furthermore, if the implementation of the task algorithm is modified, new content requirements will not entail changes to the registered interface. This reduces the fragility of the code and increases the possibility for innovative changes (without concern for breaking existing systems). Once again, even if the newly required content is not available from the existing agents, the transaction can be paused, and the new content can
be supplied or referenced by a human user. In this way, the utility of the transactions can evolve without necessarily requiring all users of the system to rewrite their code.

To further clarify the distinction, consider a scenario in which agent A1 wishes to obtain a telephone number maintained by agent A2. One can imagine that A1 is an individual searching for a friend’s phone number, and A2 is an operator, or wrapper application, with access to a telephone directory database. A request such as request-for-content(“<FullName>John Smith</FullName>”) would not have any meaning to A2 unless it understood the semantic meaning of the element “<FullName>.” The service-biased approach avoids this issue by having A2 expose an interface containing a method such as phoneNumberFromFullName(String fullName). Of course, this does not solve the underlying problem; it just recasts it as a search for the correct method (assuming that the method even exists). Furthermore, even if the method did exist, it would be extremely difficult to have agent A1 ascertain the semantics of the method at run-time.

Content-biased communication is introduced as an approach to solving the problem of run-time, or dynamic, method discovery. However, in content-biased communication, the name of the method, and indeed its very existence, has no importance. That is, the full force of the communication is conveyed through the use of content without the crutch of functional context. So, what we need is some mechanism that ascribes global meaning to elements (lexemes of the EOOTS vocabulary), such as “<FullName>.” This mechanism is the semantics and structure of the EOOTS.

5.6 Taxonomy of Content-Biased Communication Scenarios

Two complementary modes of operation shall be realized through use of the components in the global system. First, based on the discussion of C2B presented in Chapter 3, it is clear that agents will somehow be given access to user information. In
general, users will make some relevant subset of their data available for analysis by outside agents. These agents will use the data to minimize logistical burden for the users, as well as to proactively alert users to events for which they have registered interest. For instance, the clothes shopper mentioned in Chapter 3, was presented with an advertisement from a nearby clothing store because the agent representing that clothing store had access to the shopper's size data and general style preferences. This type of access to static consumer data can also be used to automate the completion of forms in circumstances, such as medical questionnaires in a doctor’s office, job applications, and countless others. In this context, the term static refers to the fact that the user’s data is not associated with a programmatic interface.

The second, and complementary mode of operation, is the initiation of multi-agent transactions. In this scenario, the user’s data will incite interest from an outside agent, but the desired activities will require an interaction between the outside agent and the user’s agent. This contrasts with the static case because the user’s data is now associated with a programmatic interface. Furthermore, the ensuing transaction may include any number of additional outside agents that provide useful services. For example, if a consumer wishes to purchase a home, there may be a requirement for additional, dynamic interactions among the buyer, seller, and broker. These interactions may be impossible to define a priori but will still be performed at runtime through the exchange of content. It is important to understand that all of the parties involved in the sale of the home understand the primary goal – namely, transfer of the title from the seller to the buyer. However, the sub-goals of each participant may still be different. Based on this fact, each participant performs its activities independently until discovering that certain required
information is missing. At that point a request-for-content is made, and the work is continued when the response is received. To clarify the distinction between these two operational modes, the following definitions of the primary operational modes of a C2B system are presented:

1. Static Interaction – a user places the relevant information in an externally accessible location; outside agents examine that data for the purpose of presenting opportunities to the user, or for saving the user the burden of manually mapping the information to the form required by the outside agent and

2. Dynamic Interaction – a user places the relevant information in an externally accessible location; outside agents initiate transactions with the user agent based on a perceived potential for mutual benefit.

In both of the operational modes discussed above, the owner of one constituent system (the user) made his data available to the owner of other constituent systems (the outside agents). The examples used to clarify the two scenarios both implied a certain form for the user’s data, namely, that the information represented some facts about the user’s preferences, possessions, or overall profile. While these examples are perfectly valid, they do not adequately convey the full potential for the system. In general, there are three forms of user data that will be made available within the global system:

1. Fact – a fact is registered in the global system to advertise some property of its user. Examples include descriptions of the user’s possessions, personal profile information, and preferences

2. Service Advertisement – a service advertisement is registered in the global system to advertise that the user can provide the described service on behalf of other agents. Examples include home-selling services, medical services, etc

3. Service Provision – a service provision is registered in the global system to advertise that the user is providing a certain service. Examples include selling a car, buying a car, etc
Some examples are necessary to help characterize the distinctions between the different formulations. Consider a user who owns an automobile. If the individual places information about that automobile in the global system, he is registering a fact. Such facts may be registered at particular locations to specify interest in receiving offers for automobile accessories specific to his make and model. Now, imagine that the user wishes to sell his car. In this case, the user will register facts about the car but will also register as a service provider. As expected, the service the user is providing is selling his particular car. On the other hand, if the individual needed help selling his car, he would look for a service advertisement. A service advertisement specifies that the advertiser will perform the service on behalf of another agent. Of course, if the auto-broker gets the contract to sell the car, then it will also register as a service provider (selling that particular automobile). It should be clear that the formulations of user information are orthogonal to the operational modes of a C2B system.

5.7 Objectives for the EOOTS Design

As discussed above, the EOOTS represents the vocabulary for our content-biased language. In reality, the structure of the EOOTS will also provide a grammar for composing the language elements. In the remainder of this chapter, we explore alternative constructions for the EOOTS grammar and examine how these alternatives affect the expressiveness and semantics of the containing language. From a high level, there are a number of desirable properties we hope to obtain from our final language structure. The following properties embody a set of guiding objectives:

1. Public Standard - no requirement of a global standardizing committee.

2. Open Standard - open specification allowing constituent system designers to freely add new concepts to the language.
3. User-Friendly - minimal thought required for determining the appropriate positioning of the new concept in the existing language structure.

4. Self-Administering - unprompted self-administration through a natural selection mechanism.

5. Explicit Relationships - clear representation of a concept’s relationship to other concepts.

6. Atomic Values - normalized, atomic data values unambiguously representing a given concept.

7. Single Source - all constituent systems map their concepts to a central type repository rather than to each other.

8. Extensible References - concepts should be given some means of referencing multiple, distinct concepts when each of those concepts make sense in the context of the concept, even if all referenced concepts are not known beforehand.

9. Differential Definition - some form of content-sharing should be possible but should not be limited to any particular relationship.

10. Contextual Independence – the designation of a concept, as well as its properties, are not affected by the relationships in which the concept participates.

11. Perspective Support – a single concept should support representation from many different perspectives.

These objectives will become clearer as each alternative EOOTS design is discussed. In the end, a single EOOTS design realizing all of the above objectives is proposed.

5.8 Exploring Potential EOOTS Designs

The English language, or more accurately an English Language Dictionary, may be considered a repository of words. Each word represents one or more concepts, as described by the definition of the word, and made real by acceptance and use in everyday dialog. In much the same way, the content-biased language (CBL) described by the EOOTS can be considered to have an underlying repository of well-known concepts. This dictionary, or Global Type Repository (GTR), represents the complete set of concepts that are globally accepted as parts of the language. The GTR (pronounced
Gator) is composed of a set of concept aggregates represented using EOOTS. Based on this description, it should be clear that a CBL is as much defined by its GTR as English is defined by an English dictionary. Furthermore, if two different GTRs were created then two different CBLs would result (just as an English Dictionary defines English and a Spanish Dictionary defines Spanish).

When a new set of concepts needs to be added to a CBL, those concepts will be represented using EOOTS. Before those concepts are registered in the GTR, they are called wild EOOTS. Wild EOOTS are not part of a CBL since they are not accessible to anyone other than their creator. If the creator of the wild EOOTS wishes to integrate his new concepts into a CBL, he must register his wild EOOTS with the GTR for that CBL. This process is called “Sewing Your Wild EOOTS.”

This section examines alternative constructions for the EOOTS. In each of the following subsections, a new model is discussed, and the shortcomings of the model are exposed. As each model improves upon its predecessor, we show how more of the objectives proposed in section 5.7 are met. Based on the experience gained from each design exercise, we present, in section 5.9, a final EOOTS design meeting all required objectives.

5.8.1 Linear Content Model

If a natural language dictionary is taken as the basis for the GTR, a reasonable EOOTS model might be structured as a set of named concepts and their associated definitions. Such a structure may aptly be named a linear content model (LCM). The definition of some concept, \( C \), in the LCM would be made in relative isolation from all other concepts in the LCM. Specifically, any other concepts in the LCM would only be related to \( C \)
insofar as they are mentioned in the description of $C$. For example, the concept “first name” could be described (using an intuitive XML syntax) as follows:

```xml
<concept name = “First Name” description = “A person’s first name”>
  <description>A person’s first name</description>
  <commonName>First Name</commonName>
</concept>
```

The XML fragment above clearly shows the definition of the concept “First Name,” and provides a human-readable natural language description of what the concept signifies. This model would certainly work, at least when dealing with first names. However, it is important to consider how such a proposal would scale to handle additional concepts such as “last name,” “phone number,” and the many components of a home address. Such a scaling is certainly possible, as one would simply create a new fragment for each additional concept and ensure that no name collisions occur. From a communication perspective, the name of the concept would clearly identify the semantics of the transmitted message. To further clarify the process, consider the more complete EOOTS model specified below (again using an intuitive XML syntax):

```xml
<AllConcepts>
  <concept name = “FirstName” id = “1” type = ”string”>
    <description>A person’s first name</description>
    <commonName>First Name</commonName>
  </concept>
  <concept name = “LastName” id = “2” type = ”string”>
    <description>A person’s last name</description>
    <commonName>Last Name</commonName>
  </concept>
  <concept name = “PhoneNumber” id = “3” type = ”string”>
    <description>A person’s home telephone number</description>
    <commonName>Home Phone Number</commonName>
  </concept>
</AllConcepts>
```

As the larger example clearly demonstrates, all new concepts are added in a linear fashion. This time, the additional concept attributes “id,” “type,” and “commonName” have been added. The “id” attribute represents a globally unique identifier (GUID) that
uniquely identifies the associated concept and helps prevent name collisions. The “type” attribute provides more detailed information concerning the representation of the associated concept and moves us one step closer to a type-aware model. Finally, the “commonName” attribute provides a human-friendly name for the concept for use on forms or other user interface media. Given the example above, the message reply-with-content(“FirstName id = “1”>John</FirstName>”) would unambiguously relate the message to transmission of some individual’s first name. While we do not suggest this approach as the ultimate solution, it does have some attractive features. Observing and understanding these features, and also identifying the shortcomings, will help move us towards a more robust solution.

The first positive feature of the LCM is that it does not require a centralized standardizing committee. There is no technical reason why constituent system developers could not simply add new concepts as they saw fit. This is a key feature for the success of any candidate proposal. Otherwise, a constituent system developer would be forced to wait for the standardizing committee to approve additions whenever the developer needed to represent a new concept. However, the lack of standardization does have its downsides. For example, it is likely that conflicts (or redundancy) would arise in the creation of new concepts. It is our belief that such problems would only detract slightly from the general utility of the system. Suppose that both “lname” and “lastName” were bound to the same semantic concept (described to impart the same meaning to the human reading the associated “description” tags). In this situation, the two concepts would be interpreted as being independent, though in reality they are identical. Should this happen too frequently, the system would begin to lose its efficacy,
unless constituent system developers made a constant effort to map new redundant tags, either to each other, or to the correct concept in their local systems. While this may seem impractical on the one hand, on the other hand, it can be viewed as a form of natural selection. Consider what may transpire if a clumsy, discourteous constituent system designer added a redundant concept to the GTR. If no other constituent systems ever used this new concept, or even mapped their equivalent concepts to this new concept, then the discourteous designer would be forced either to retract his mistake or to live without the interoperability benefits provided by the system.

Another attractive feature of the linear content repository is that little time needs to be spent deciding where to place a new concept. Since concepts are simply added in a linear fashion, the constituent system designer just adds the appropriate fragment. However, this positive aspect also brings with it a set of more serious problems. First, the linear approach fails to adequately represent the notion of context. Consider the addition of the concept “rim.” In and of itself, this concept actually has no useful meaning. The tag does not tell the user if this rim belongs on a car, a bicycle, or a hat. Of course, the “description” element may clarify the intended usage by stating that the “rim” tag represents rims on automobiles. However, this implies that separate tags are required for a car’s rim, a bicycle’s rim, and a hat’s rim. Given the functionality of the “id” attribute, and the assumption of well-written concept descriptions, such difficulties would certainly not prove insurmountable. However, consider now the addition of the concept “car.” The name of this tag does not specify if this car is owned, leased, or being repaired. Of course, the “description” tag may again be used to specify that the element, “car” designates the set of cars belonging to some individual. But, we are still left ignorant as
to the identity of this individual. Furthermore, if the “car” tag represents cars presently owned, then must there also be a tag, such as “ownedCar,” specifying cars previously owned but no longer owned? These examples clearly indicate that the linear approach will not meet the requirements of a basis for a CBL at least not in any practical sense.

Before completely dismissing the linear representation, it is useful to highlight one additional deficiency. The “description” of a “car” concept may specify that the intended usage of this concept is to present a string such as “1999 Chevrolet Corvette” reflecting the fact that the owner (however his identity may be ascertained) presently owns the Corvette. Such a value may be useful in some circumstances, but it is likely too complex to satisfy the majority of applications. Simply obtaining the year of the automobile would require parsing the string value. This raises a very important point, namely that the values supplied in the content should be defined with an eye towards atomicity. This observation coincides very closely with the notion of first normal form in the study of relational databases. As we shall see, some of the most desirable properties for an EOOTS model are intimately related to the theory of relational systems.

5.8.2 Hierarchical Content Model

Perhaps the most notable shortcoming of the linear content model is its inability to adequately represent context. The utility of context is not to be underestimated. For instance, consider the “rim” mentioned in the previous section. By simply stating the context for discussion of the rim, a human speaker clearly distinguishes the target concept from any related concepts and also brings a number of associated properties of that concept to the forefront of the discussion. These two consequences of using context are important enough to warrant recognition in the following pair of definitions (see section 5.8.4.3 for a third):
1. Concept Discrimination – the use of context to distinguish between two or more concepts traditionally referred to using the same lexeme and

2. Concept Restriction – the use of context to delineate a particular, relevant set of properties of a concept.

Returning to the “rim” example, concept discrimination can be used to differentiate an automobile rim from a bicycle rim. It should be clear that both types of rims are indeed rims, and that the requirement for contextual disambiguation stems from the lack of distinct vocabulary elements in the English Language and not from some metaphysical entwining of the two concepts. That is, if one traditionally called an automobile rim an “x” and a bicycle rim a “y,” then the need for concept discrimination would be nullified. However, in this case, the linear content model would have sufficed and there would be no need for additional investigation. Alas, there is no natural language that has a distinct name for every distinguishable concept, and it is common practice to use ever-narrowing contexts to provide the necessary disambiguation.

In concept restriction, context is used to distinguish many different perspectives on a single concept. For example, given an automobile rim, a packaging company may be interested in properties of the rim including its weight, diameter, and depth. On the other hand, a consumer of the rim may be more concerned with the style, price, and manufacturer of the rim. By supplying a context, such as “packaging” or “consumer sales,” a different set of properties can be isolated. Such partitioning is very important since the complete set of properties for any given concept can be extremely large, and only a very small subset is generally relevant in any given situation.
The immediate solution for adding context to the EOOTS is to shift to a hierarchical content model (HCM). In this case, if a rim is an automobile rim, then it will appear beneath automobile in a hierarchical collection of concepts. To help clarify this design, the diagram in Figure 5-1 is presented. Note, we have intentionally moved away from a textual (XML) representation of our models and proceeded to use a more flexible, and clearer, graph-based model. Though a final model for the EOOTS is presented at the end of this chapter, it is not until Chapter 7 when we once again consider conversion to a textual representation.

The graph in Figure 5-1 demonstrates some clear improvements over the linear content model. First, the appearance of the rim concept beneath car and wheel seems to provide adequate context to tacitly imply an interpretation of rim as an automobile rim. More importantly, the location of the rim concept in the graph, the path from person down to rim, provides a unique means of identifying the associated concept. Furthermore, the path from person to rim can be distinguished from the rim node itself. The former is a description of a concept in context, while the latter is simply an isolated description of a concept.
While the HCM does provide support for concept discrimination, it has no means for handling concept restriction. This is definitely a serious shortcoming of the model, but it is not the only one. Referring back to the illustration in Figure 5-1, it appears that some relationship is implied between the concepts of person and car. The most likely candidate for this relationship is ownership, but it is important to realize that this relationship is not present and cannot be assumed. Furthermore, it seems that some of the concepts under car are properties of the car, while others are components of the car. In other words, the edges in the graph have no single, discernable meaning. This shortcoming is quite serious and is tackled in the following section.

5.8.3 Edge-Labeled Hierarchical Content Model

The previous section closed after identifying an important limitation of the hierarchical content model. In particular, the edges in an HCM graph did not clearly assign any particular relationship between adjacent nodes.

![Figure 5-2. Edge-labeled hierarchical content model](image)

To address this problem of missing relationships, the edges in an HCM graph can be decorated with explicit relationship identifiers. The resultant representation is called an
edge-labeled hierarchical content model (el-HCM) and is illustrated in Figure 5-2. The el-HCM definitely affords its user far greater expressiveness and clarity in describing a set of concepts. Finally, concepts can be placed in context, and the relationships between those concepts can be made explicit. However, as the remainder of this section details, the el-HCM is still fraught with numerous problems.

5.8.3.1 Properties of a relationship

Once edges are labeled with relationships, it becomes possible to clearly specify that a person owns a car. This relationship is modeled in Figure 5-3. Consider now that we wish to identify the dates of ownership. In this case, we are trying to associate certain properties with a relationship. However, since a relationship is represented using an edge label, there is no facility for attaching properties. One could certainly draw edges emanating from edges and pointing to nodes, but this destroys the regularity of the graph structure. Another possibility is to add additional child nodes to the car node. These new nodes, representing the start and end dates of ownership, are illustrated in Figure 5-4. The problem with this solution is that we have now added properties to car that truly have nothing to do with the car concept. In fact, under the constraints of the el-HCM there is no adequate solution, thus highlighting the first major problem with the model.

5.8.3.2 Multiple relationships between concepts

Another issue that arises when modeling relationships between concepts is that the same two concepts often have multiple relationships between them. Consider the situation in which a person not only owns cars but also customizes them. In this case, the single “owns” relationship between “car” and “person” is not sufficient. Two alternatives
Figure 5-3. Edge-labeled model enables expression of the car ownership relation.

Figure 5-4. Acquisition date is not a true property of car.

Immediately present themselves. First, we can add another element under car to specify whether this car was owned or customized. This option is illustrated in Figure 5-5 but
should clearly be discarded. Since the edge between person and car is already labeled “owned,” it makes no sense to override that information at a lower level in the tree. In fact, this would completely eliminate any reasonable interpretation for the edge labels at all.

A second option is to add another edge between person and car (see Figure 5-6). This new edge is labeled “customizes” and points to cars that the person has worked on. To see this, recall the discussion of paths in section 5.8.2. The path person/(owns)/car designates cars owned by the individual, while the path person/(customizes)/car specifies the set of cars on which the person has worked. This addition of multiple edges between nodes suggests the need for a new model designation, but to do so would really only serve as an exercise in pedagogy.
5.8.3.3 N-ary relationships

It often occurs that a relationship exists between more than two concepts. For example, in the text “An Introduction to Database Systems” by C.J. Date, a hypothetical schema is presented describing the relationships between suppliers, parts, and projects. In general, a supplier supplies parts to a project. Consider the following three binary relationships:

1. suppliesWhat - relationship between suppliers and parts – specifying which parts a supplier supplies,
2. suppliesTo - relationship between suppliers and projects – specifying to which projects a supplier supplies parts, and
3. usesWhat - relationship between projects and parts – specifying which parts a project uses.

Given only the three relationships presented above, it would be impossible to determine which supplier supplied a particular part to a specific project. What is needed is a ternary relationship, “suppliesWhatToWho,” that relates all three concepts simultaneously. Unfortunately, it is impossible to model such relationships directly using an edge-labeled graph. The problem is that an edge, by its very nature, can only relate two concepts at a time. One solution to this problem is to use anonymous nodes to simulate the required behavior. For a more detailed description of this approach, see the
RDF standard presented by the W3C. For our purposes, we consider this shortcoming to be another problem with the el-HCM.

5.8.3.4 Cyclical relationships

In many situations, an instance of a concept may have a relationship with other instances of the same concept. For example, Russian Dolls, directories in a file system, and atoms in a molecular system, all maintain relationships to other instances of their associated concept. Furthermore, it is possible for a single instance of a concept to have a relationship with itself. To help elucidate such circumstances, a number of diagrams are presented in Figure 5-7. As illustrated in the figure, such relationships can be modeled as cycles in an el-HCM graph. Figure 5-7(a) models a hypothetical mechanism in which a gear turns a spindle, and a spindle then turns a gear. In Figure 5-7(b), a file system is modeled demonstrating the recursive contains relationship between directories. Figure 5-7(c), shows a person who owns a car but then also shows an edge from the car to the person labeled with the relationship “owned-by.” Finally, in Figure 5-7(d), the person-car model is extended to illustrate a larger cycle in which a person “owns” a car, the car “has” a wheel, and thus the wheel is “owned-by” the person.

We have included both Figure 5-7(a) and Figure 5-7(b) to demonstrate that cycles may occur as self-cycles in the case of Figure 5-7(a), or as longer cycles as shown in Figure 5-7(b). In either case, a contextual difficulty arises since there is no longer a fixed length for each concept in the graph. For example, consider the following paths:

“filesystem/(contains)/directory/(has)/name” and

“filesystem/(contains)/directory/(contains)/directory/(has)/name”
Both paths are valid in the el-HCM graph of Figure 5-7(b). However, one must question whether each represents the same concept or instead represents a new and different concept. A solution favoring the former interpretation rests closer to intuition. Specifically, a directory is a directory, regardless of its level in the file system hierarchy. Given this interpretation, it is fair to state that all paths of the form “filesystem/[(contains)/directory/] + (has)/name” represent the same overall concept. Of course scenarios may develop where smaller cycles occur in larger ones, and the expressions quickly become more complicated.

The graph in Figure 5-7(d) motivates discussion of another problem related to the existence of cycles. Here, the relationship stating that a person owns a car is modeled as an edge. Intuitively, if the person owns the car, and the car contains a wheel, then the
person also owns the wheel. However, since there is no relationship between the relationships, this type of distribution cannot be assumed. That is, the ownership relation between person and car does not necessarily entail an extended ownership relation between person and any other concepts that fall beneath car in the graph. One possible solution is to make the wheel-ownership relationship explicit by adding another edge between person and wheel (as shown in Figure 5-7(d)). In the illustration, the addition of this new edge creates a cycle, but that is certainly not a requirement. In fact, the existence of the cycle generates further problems in the interpretation of the graph. It is logical to assume that the person who owns the car is the same person who owns the wheel. However, making this assumption reaches into dangerous territory because it imposes a restriction on the values of the instances that conform to this type.

At this point, we have identified two problems with allowing cycles in a graph. First, the existence of an infinite number of paths, different only in the number of occurrences of a repeating group, creates problems in defining a path-to-concept mapping. Second, the addition of certain cycles to the graph induces constraining assumptions regarding the validity of concept instances. The second issue appears to be tractable, but the first problem is simply intolerable. The purpose of the EOOTS model is to represent concepts (data types in context), not data. Therefore, if two concepts are equivalent, then it makes no sense to have two different representations for them (it) in the concept graph. In reality, this discrepancy arises because we are violating the foundation axiom of Zermelo-Fraenkel set theory. The axiom states that every non-empty set has a foundation member and is formally represented as follows:

$$\exists y (y \in x) \Rightarrow \exists y (y \in x \land \neg \exists z (z \in x \land z \in y))$$
The foundation member, y, must not be defined in terms of any other member of the set. In much the same way, we will now disallow cycles and avoid the possibility of a concept being defined directly, or indirectly, in terms of itself.

5.8.3.5 Reference nodes

In the previous section, we highlighted a number of problems associated with allowing cycles in el-HCM graphs. If cycles are indeed prohibited, then some alternative mechanism must be presented to handle the common occurrence of self-referencing concepts. The graphs presented in Figure 5-8 illustrate one possible option. For each graph in Figure 5-7, a new graph, with cycles removed, is provided in Figure 5-8. In each case, the cycles present in the original model have been replaced with a new node type called a reference node. A reference node (shown in the diagrams as a dotted oval) is akin to the concept of a foreign key in a relational database. That is, a reference node designates a relationship between two concepts by providing a pointer to a related concept.

Where the reference node appears in a graph, it could be replaced with an edge to the type of node it references. However, the meaning of the cycle that may occur as a result of adding that edge differs from the interpretation of the reference node. The cycle creates a new path in the graph and implies that some previously seen concept may also appear at this “lower” level. The reference node, on the other hand, points to a concept and designates that the concept has some relationship to one of the reference node’s ancestor concepts. The important difference is that the cycle moves the concept to a different position in a path, thereby changing its context, while the reference node simply implies the existence of a relationship between concepts.
It is useful to note that reference nodes are not limited to representing relationships between concepts that fall on the same path. In fact, reference nodes can be used to designate relationships between any two nodes in a graph, even if creation of an edge between those nodes would not result in a cycle. Realizing this fact motivates us to explore an alternative to the edge labeled model that we have been discussing thus far. This is a promising sign, since it should be clear at this point that an edge-labeled model has far too many problems to ever yield a practical solution.
5.8.3.6 Removing the edge labels

The previous section concluded by stating that the el-HCM was impractical due to its dependence on edge labels. The problems associated with edge labels are summarized in the following list:

1. Does not support properties of a relationship.
2. Does not support n-ary relationships.
3. Does not support role-association for concepts in a relationship.
4. Violates the principle of contextual independence.

The first two problems listed above were described in Sections 5.8.3.1 and 5.8.3.3, respectively. The second problem was not discussed but is reasonably easy to comprehend. Consider the binary “owns” relationship between person and car. Two different representations of this relationship are conceivable. In one alternative, an edge would source from person and point to car, with a label such as “owns.” Another possibility is that the edge sources from car and points to person. In the second alternative, the edge could be labeled “ownedBy,” but could also be labeled “owns.” The point is, the two concepts are playing different roles in the relationship, and there is no explicit means of identifying those roles. Changing the name of the edge-label may provide some clarification, but it is certainly not the optimal solution.

The fourth problem listed above is somewhat less intuitive. Consider the graph shown in Figure 5-8(a). The fact that there is a “turns” relationship between spindles and gears is clear, but unfortunately it is interwoven with the hierarchical structure of the graph. The discrepancy arises because we are violating a principle we call contextual independence.
Principle of Contextual Independence: The designation of a concept, as well as its properties, are not affected by the relationships in which the concept participates.

Referring to the graph in Figure 5-8(a), we can now state that the interpretation of an edge was overloaded to represent both contextual designation and relationship. Relationships, in general, are orthogonal to the concepts they connect. To illustrate this point more clearly, we have provided the three graphs shown in Figure 5-9. This figure presents three models of a molecular system. The goal of each model is the same, namely the representation of molecular systems. For example, a DNA molecule (or
molecular system) is composed of four primary sub-molecules (Cytosine, Thymine, Adenine, and Guanine), which are in-turn composed of atoms. The models are created to represent molecules and atoms, as well as three relationships that may occur between them. These three relationships include molecule-to-molecule bonds (between sub-molecules), atom-to-atom bonds (within a sub-molecule), and molecule-to-atom containment specifying the set of atoms contained by a particular sub-molecule.

The graph shown in Figure 5-9(a) uses edge labels with cycles to model the required concepts and relationships. In Figure 5-9(b), the same information is modeled with the cycles replaced by reference nodes. Finally, in Figure 5-9(c), the edge-labels are removed, resulting in a graph with concepts on one side and relationships on the other. This final iteration clearly shows a separation between the use of edges for context and the representation of relationships. In fact, upon further examination of the graph, it should be clear that the set of concepts being modeled are completely independent of whatever relationships may occur between them. Thus, we have achieved our goal of contextual independence.

The model shown in Figure 5-9(c) also solves the remainder of the problems mentioned at the start of this section. First, it is now possible to represent properties of relationships. This can be seen in the “type” node beneath the “bond” node representing whether the bond is covalent, ionic, or some other type. The problem of n-ary relationships is also solved, since it is now possible to place any number of related nodes beneath a relationship node. Finally, role-association is also supported. Beneath the “contains” relationship, “src” and “dest” nodes are used to designate the container and the
contained. In the case of “bond,” the relationship is symmetric, and no special
discrimination of roles is required.

This new model represents a major step forward in the design of a viable EOOTS
structure. However, there are still a few more improvements that need to be made.
These additional features are discussed in the remainder of Section 5.8, and then a final
model is presented in Section 5.9.

5.8.3.7 Comparison with the relational model

It is interesting at this point to take a step back and look at our current model from
another perspective. Consider the relational database schema presented in Figure 5-10. In
the schema, each of the nodes beneath RelatedConcepts in Figure 5-9(c) has been
modeled as a database table. Furthermore, relationship tables have been established for
each of the children under the Relationships node. However, since a foreign key
constraint in the relational model can only point at one table, four relationship tables were
required to completely represent all relationships.

This analogy raises two interesting issues. First, our model of communication
metadata is clearly related to a widely accepted model of storage metadata. The most
striking aspect of this relationship is that both models gain advantage by separating
concepts from relationships. Unfortunately, in the relational model many concepts are
often aggregated into a single table. These concepts are considered properties of
whatever concept the table represents, though no explicit relationship is made between
the two. The second point of interest relates to the necessity of four relationship tables in
the schema. In reality, this requirement stems from the relational model’s inability to
represent an “is-a” relationship. This issue is examined in more detail in Section 5.8.5
where we discuss content-sharing. The relational model is nearly perfect for representing
storage metadata, and rather surprisingly, it is also a very strong candidate for representing communication metadata. Though the form and utilization of the two metadata models differ markedly, it is definitely worthwhile to keep their similarities in mind.

Figure 5-10. Relational schema

5.8.4 Relationships, Properties, and Globally Unique Identifiers

Up to this point, the notion of a concept has been rather vague. In fact, the term concept has been used interchangeably to represent an entity, as well as each of its individual properties. As a result of this informality, concepts such as “car” and “birthdate” may legitimately be placed under a parent concept, “person,” as siblings of equal stature. Such a model may be considered inappropriate based on a feeling that there is some fundamental difference between the two concepts warranting a more formal and explicit distinction. In this section, we examine a number of distinctions that may be drawn between properties.
5.8.4.1 Intrinsic and extrinsic properties

Returning to the comparison between car and birth-date, we can find two major differences. First, the birth-date of an individual remains the same for all time, whereas a car may be bought or sold at will. Furthermore, the conceptual notion of a car may be considered to lie outside the concept of a person, whereas birth-date is an inherent property of the individual. This distinction between properties of a concept is best expressed using the following pair of definitions:

1. Intrinsic Properties – the attributes of a concept that occur as a result of observing the concept in the absence of all external relationships - the inherent or natural attributes of a concept and

2. Extrinsic Properties – the attributes of a concept that occur as a result of observing the concept’s external relationships – the external or disconnected attributes of a concept.

In the definitions presented above, the distinction between intrinsic and extrinsic properties is based on the notion of bounded systems. For example, when one considers a person, the intrinsic properties of the person are those properties that are discernable when viewing only the person. Examples of such properties include birth-date, race, and eye-color. Extrinsic properties require the observer to look outside the bounded system of the concept. Such properties include material possessions, friends, and experiences. It would appear that the distinction between intrinsic and extrinsic properties is quite objective, but unfortunately this is not the case.

Consider the “make” property of an automobile. While this property may initially seem intrinsic to the car, it may also be considered an extrinsic property corresponding to the name of the manufacturer. That is, the “make” of the car is not a property of the car at all, but is instead the result of a “manufactures” relationship between the car and the company that built it. A more definitive example demonstrating the fuzziness of any
distinction between intrinsic and extrinsic properties can be found in the notion of name. A person’s name is almost certainly an intrinsic property of the individual. However, consider that a name can be changed (and often is after marriage). Furthermore, a name is really just an artifact of society and has no real correlation to the individual. Based on these observations, there does not seem to be any clear and precise methodology for deciding whether a property is intrinsic or extrinsic. However, there is a certain unshakable comfort associated with retaining the distinction. After all, it exists in the relational model where some properties of an entity are represented as fields, while others are given their own table.

Rather than immediately discarding the distinction, it is useful to give it some further consideration. In the theory of relational databases, properties that would create a repeating group are always given their own table. Thus, since a person may own more than one car, it makes sense to consider car a separate concept and define it in a separate table. By the same logic, since a person may have only a single birth-date, birth-date is not a separate, independent concept and need not be given its own table. Another possible distinction that can be drawn between car and birth-date is that the former may have many sub-components and properties of its own, while the latter is essentially an atomic value. With these distinctions in mind, it is interesting to consider the concept of height. Certainly, an individual’s height does not remain constant for all time, so supposedly it should be modeled as a distinct concept. On the other hand, height is an atomic value, so it does not really need its own table. Furthermore, for many applications, height is not considered variable, and only a single value is needed (e.g., a driver’s license).
The struggle to determine when a property should be promoted to the stature of a concept is really a subjective issue based on context. In any given context, the perspective chosen by the application induces a view of reality as it exists at some unique level of granularity. In other words, the application exists in, or models, a particular physical-behavioral unit and, as a result, sees the world through an associated granular partition. When the elements of a driver’s license are defined, the granularity requires that height only be considered at one particular moment in time. In the same context, the granularity required for eye-color associates the property with a person, not with his eye, nor its iris. Thus the question of intrinsic versus extrinsic properties is an issue of granularity and will differ based on the requirements of each individual application. As would be expected, applications that view the universe at the same level of granularity are more likely to communicate and would also have a better chance of doing so.

5.8.4.2 Globally unique identifiers – more useful distinction

In the previous section an effort was made to define which concepts may legitimately be placed beneath other concepts in a hierarchy. The section concluded by stating that the choice was purely subjective and depended on the requirements of the particular application. Thus, if it is convenient to designate car by placing it in a path somewhere beneath person, then that is exactly the action that should be taken. This makes sense if we recall that the primary goal in designing an EOOTS model is assigning each concept a unique, recognized position in some structure (i.e., a path in a graph). Furthermore, no effort can, or should, be made to decipher implicit meaning based on the chosen position of the concept in the structure. That is, we are not trying to model constraints on data by placing concepts at a certain location. Rather, the intention is to partition reality at some convenient level of granularity and then provide unique and intuitive designations for
each of the concepts present in that reality. This is the very essence of communication metadata and is based on the notion of granular partitions introduced in the study of quantum mereotopology.

That said, there is still one particularly useful distinction that can be drawn between the set of concepts in an application. For some concepts, the instances of the concept must be distinguished from one another; for other concepts, this is unnecessary. For example if there are one thousand pieces of fruit in a truck, it may be useful to detail the type of fruit, but it will almost certainly be unnecessary to distinguish each individual piece of fruit from all others. On the other hand, if we have one hundred patients in an emergency room, it is absolutely imperative that each one is uniquely identified. Once again, the decision concerning which concepts receives unique identities and which do not is specific to each application. However, within an application, this distinction can be used to dictate placement in the hierarchy.

When modeling a concept, there are three alternatives related to selecting an appropriate identifier. These three concept identification options are outlined in the following list:

1. Globally Unique Identifier (GUID) – distinguishes each instance of the concept from all other instances of the concept in the universe,

2. Relatively Unique Identifier (RUID) – distinguishes each instance of the concept from all other instances of the concept present in some particular context, and

3. No Unique Identifier (NUID) – no distinction is made between instances of the concept.

A GUID is useful in circumstances such as the emergency room mentioned above. In such cases, each instance of a concept must be distinguished from all other instances of the concept through time and space. To make the utilization of GUIDs truly effective, it
is imperative that only one GUID be assigned to the same instance, regardless of the application. This clearly distinguishes a GUID from a primary key in a relational database. The former is unique everywhere, while the latter is only unique within the context of some particular data store. Realizing this goal is not trivial, we will raise the issue again when discussing potential architectures for a global system.

A RUID is used when global uniqueness is not a requirement. For example, when discussing eyes in the context of a person, there is no need to associate each eye with a GUID. Rather, it is useful to distinguish the left-eye from the right-eye. Such a distinction would obviously be important in medical applications and can be accomplished by simply designating which eye is being discussed. It is important to realize that the concepts left-eye and right-eye are the same and are distinguished only through use of a RUID associated with the concept eye. In a situation such as this, it clearly makes sense to put the eye concept under person since the person provides the context on which the RUID depends. In other words, we are not modeling left-eyes and right eyes; instead, we are modeling this person’s left-eye and this same person’s right-eye.

The final option, NUID, provides for some of the more interesting modeling examples. For instance, in the case of the fruit-carrying truck, it makes perfect sense to model fruit under truck. In almost all other circumstances there is really no relationship between these two concepts. However, in this case, specifying the attributes of the cargo beneath the carrier may yield the most intuitive model. The following outline helps clarify the relationship between the three concept identification options and their interpretations and proper use in a hierarchy:
1. Globally Unique Identifier (GUID):
   a. generally placed as a direct descendent of the root concept of the graph,
   b. all concepts beneath the GUIDd concept are interpreted in the context of the GUIDd concept, and
   c. if a GUIDd concept appears beneath another GUIDd concept, then the second GUIDd concept resets the context of the path and is related to the original GUIDd concept only by means of some explicit relationship.

2. Relatively Unique Identifier (RUID):
   a. almost always placed beneath a GUIDd concept and
   b. when under a GUIDd concept, the RUIDd concept is interpreted in the context of its most recent parent GUIDd concept.

3. No Unique Identifier (NUID):
   a. may appear anywhere in the graph and
   b. when under a GUIDd concept, the NUIDd concept is interpreted in the context of its most recent parent GUIDd concept.

5.8.4.3 Multiple graphs

The rules governing concept identification in a hierarchy will become clearer when discussed in the context of our final EOOTS model (Section 5.9). Presently, it is tempting to suggest that RUID and NUID concepts beneath a GUID concept are intrinsic properties of the GUID concept, while GUID concepts beneath another GUID concept are extrinsic properties. Unfortunately, this is an oversimplification and may lead to incorrect conclusions. To clarify this point, we complete the list started in Section 5.8.2 by adding a third and final use for context (extension of list in Section 5.8.2):

1. Concept Discrimination – the use of context to distinguish between two or more concepts traditionally referred to using the same lexeme,

2. Concept Restriction – the use of context to delineate a particular, relevant set of properties of a concept, and
3. Concept Positioning – the use of context to provide a convenient basis for conversation concerning a concept.

This third use of context, concept positioning, relates to the location of a concept in a hierarchy of concepts. Concept positioning uses context as an argument for placing one concept beneath another. That is, if for some particular application it is normal to speak of one concept only in the context of another then it is acceptable to place the latter beneath the former within a hierarchy. In this way, any concept may be placed under any other concept if it makes sense for the application. However, it is imperative that the positioning of a concept is understood to have no effect on the nature of that concept. Thus, if fruit is modeled under truck in one application, and under elephant in a different application, it is still possible that both versions of fruit refer to the same thing. If that is the case, then context has been used for concept positioning. Otherwise, context was used for concept discrimination, and the fruit under elephant is an entirely different concept from the fruit under truck (recall “car/rim” versus “bicycle/rim”).

The distinctions made between the uses of context, and how they are related to concepts in a graph, require one final point of clarification. In particular, the existence of multiple graphs must be presented. Generally, there is one EOOTS graph per constituent system. This graph models all concepts and relationships relevant to the physical-behavioral units observed by the applications in the constituent system. However, different constituent systems see the world from different perspectives, and consequently, model the world in different ways. As described at the beginning of Section 5.8, the GTR is composed of a set of EOOTS graphs. Each graph models the world from a different perspective, but that does not mean that they model disjoint sets of concepts.
Thus, there must be some way to map these different perspectives enabling different constituent systems to communicate with each other.

With this in mind, the distinction between concept positioning and concept discrimination can be made more explicit. Concept positioning states that in two (or more) different graphs, a single concept can be placed in two different locations without affecting its meaning. Concept discrimination states that in a single graph, the same name can be applied to different concepts, and the distinct location of those concepts distinguish one from the other. Furthermore, it is unacceptable to represent a single concept, in a single graph, using two different locations. So, within a single graph, each concept may appear only once, and two nodes with the same name, but in different locations, are guaranteed to represent different concepts. On the other hand, in different graphs, a single concept may be represented in different locations, and signified using different names, and yet still be considered the same concept. Now, we can restate the definitions for the three uses of context (extension of list in section 5.8.2):

1. Concept Discrimination – in a single graph, the same name can be applied to different concepts, and the distinct location of those concepts distinguish one from the other,

2. Concept Restriction – the use of context to delineate a particular, relevant set of properties of a concept, and

3. Concept Positioning – in two (or more) different graphs, a single concept can be placed in two different locations, and given two different names, without affecting its meaning.

5.8.5 Adding Content Sharing into the Model

In the realm of programming language type systems there are three primary techniques for handling the abstraction of common or reusable functionality. Each of these approaches is illustrated in Figure 5-11. In Figure 5-11(a), aggregation is used to allow
the Tiger class to expose the generic functionality of the Animal class. In this model, the functionality implemented in the Animal class can be accessed as if it were a part of the Tiger class (though a request for the proper interface may be required). In Figure 5-11(b), the Tiger class contains, or delegates to, the Animal class, by wrapping the Animal methods with corresponding Tiger methods. In this case the Tiger class may specialize the Animal functionality by adding additional code in its version of the methods. Finally, Figure 5-11(c) illustrates the standard inheritance relationship as seen in languages such as C++ and Java. In general, these relationships are applied to implement both differential definition and polymorphism. However, when considered in the context of the EOOTS, these relationships are not particularly useful. Remember, the EOOTS models only content, not methods.

![Diagram](image)

Figure 5-11. Three types of content sharing

What is required for the EOOTS is a mechanism that allows different concepts to be referred to in a single, uniform fashion and some means of sharing or reusing content across those different concepts. That is, instead of implementation-inheritance or interface-inheritance, we require a new relationship called content-sharing. If the term *inheritance* is consistently replaced with the term *sharing*, an orthogonal relationship
between content, implementation, and interface is uncovered. This relationship is illustrated in Figure 5-12. It is important to understand that opportunities for sharing (whether content, implementation, or interface) are totally independent of the “is-a” or containment relationships. In fact, the only reason differential definition and the “is-a” and containment relationships are considered inseparable is that most modern object-oriented languages use inheritance to model all of them simultaneously. By separating these different relationships, we find that there is no longer a requirement for common parentage to take advantage of the benefits of content-sharing. However, there is also no guarantee that concepts participating in an “is-a” relationship will share any content. As shown in Section 5.9, this is exactly the behavior desired for modeling an EOOTS structure.

Figure 5-12. Separating implementation, interface, and content sharing.

### 5.9 Workable Model

The analysis described throughout section 5.8 helped to clarify many desirable properties of an EOOTS model. Using that work as a basis, it is now possible to introduce a new, more powerful model, supporting all of the features identified. However, just as with any model used for representing reality, the burden of proper
utilization still rests on the shoulders of the developer. An EOOTS model, like a model for an object-oriented type system, provides developers with the tools necessary to represent their application domain. However, just as with an object-oriented model, an EOOTS model cannot ensure that developers use the tools properly or effectively. As the model presented in this section sees more use, it is our hope that design patterns will emerge to direct developers in effective design.

5.9.1 Perspective Domain Graphs

As alluded to many times throughout this chapter, an EOOTS model is a software-oriented structure for encoding the communication metadata related to a physical-behavioral unit (PBU), or more specifically a granular partition corresponding to some PBU. A number of different structures were considered, but up to this point, no single structure has satisfied all requirements. In this section, a new model called the Perspective Domain Graph (PDG) is presented. A PDG represents a subset of the universe as seen from the perspective of a physical-behavioral unit. The PDG encodes the important details of a PBU in such a way that the details can be easily utilized as topics of discourse. A PDG is a directed acyclic graph (DAG) with four node types:

1. **Concept Node** - Represents an identifiable entity in the PBU.
2. **Relationship Node** - Specifies a relationship between sets of concepts.
3. **Reference Node** - Refers to the subjects or concepts that take part in some relationship.
4. **Subject Node** – Expresses similarity across a set of concepts.
Figure 5-13. Perspective domain graph

Consider the simple example of car ownership illustrated in Figure 5-13. In this PDG, two primary concepts are identifiable, the car and the owner. For this example, we will assume the owner is a person, though that need not always be the case. On the right hand side of the figure, a legend clearly designates the different levels of the graph. The source (root) node of a PDG is always a concept node and is therefore situated in a concept level (CL). Following all but the final concept level, there will always appear a Relationship level (RL), Reference level (XL), and Subject level (SL) in that order. As expected, an RL only contains relationship nodes; an XL contains only reference nodes; and an SL contains only subject nodes. Furthermore, the sequence of levels (CL, RL, XL, and SL) repeats as many times as necessary to represent all required concepts. For this particular example, the subject level is empty, but the dotted box (explained later) shows where such nodes would appear.

Edges in the graph are unlabeled. This is because all relationships are expressed as nodes in the relationship levels. In the case of Car-Ownership, there is only one user-
defined relationship, <owns>. The <owns> relationship has two reference nodes as children. Each reference node is identified by a name and specifies the content-type (discussed later) of its referent. Thus, the <owns> relationship clearly relates an owner-person to an owned-car.

The <ctx> node is a special relationship that is always present. In addition, there is always one child reference node under the <ctx> node for each concept at the following concept level. The child reference nodes of <ctx> have no name and no referent concept-type. The <ctx> node is best explained by equating it to the phrase “in the context of.” Thus, <person> in the context of <Car-Ownership> accurately signifies the <person> node in the graph.

Obviously, the example above is rather trivial. In general, there are attributes associated with some of the concepts, and often it may be useful to refer to a set of concepts rather than only one. The PDG shown in Figure 5-14 illustrates a situation in which both of these requirements are supported.

Once again, the root node is Car-Ownership. Now, however, the person may own a car or a truck. To address this possibility, the model allows a reference node to point at a subject node. In this case, the subject node, <vehicle>, points to <car> and <truck> signifying that both of those concepts are examples of a vehicle. Since both cars and trucks are vehicles, it would be advantageous to enable the sharing of attributes. This is
Figure 5-14. Car-ownership PDG

accomplished through use of a named <ctx> node. The node, <ctx S=vehicle> is used to specify the content that shall be shared between <car> and <truck> when they are considered vehicles. Notice that the <towingCapacity> attribute of the <truck> is not included under the vehicle context.

In general, there are two types of edges in the graph, intensional edges and extensional edges. Extensional edges connect subject nodes to concept nodes or other subject nodes.
and are best understood as representing an “is-a” relationship. All other edges in the graph are intensional edges and can be interpreted as signifying an in-the-context-of relationship. The following list outlines all possible edge types:

1. Concept-to-Relationship,
2. Relationship-to-Reference,
3. Reference-to-Concept,
4. Reference-to-Subject,
5. Subject-to-Concept, and
6. Subject-to-Subject.

Notice that the attributes of a concept are modeled themselves as concepts. Starting from the root concept, any other concept can be reached using a path containing only <ctx> nodes (and the associated reference nodes). That is, relationship and subject nodes can be completely avoided. Given this fact, it is clear that any concept must make sense in-the-context-of all its ancestor concepts since that is the relationship implied by all of the edges and the <ctx> nodes on the path. Given this concept-to-concept relationship, it should be clear that the descendant concepts of a concept are not limited to properties but may be any concept that makes sense in that context. When the concept is a property-like attribute with a simple type, rather than a set of child nodes, it appears as a leaf node in the graph. In the example above, the <name> concept is a simple attribute (possibly with type string). However, if more detail were needed, <name> could itself have children such as <first> and <last>.

It is important to realize that sub-classing and differential-definition have been separated. The former is represented using subject nodes, while the latter is realized through user-defined <ctx> nodes with an “S” attribute set equal to some subject. It
should be noted, however, that the value of the “S” attribute need not correspond to an existing subject node. Equating the “S” value to an existing subject node provides an inheritance-like relationship between concepts, something we refer to as content inheritance. In other situations, when the “S” value has no correspondence, it serves as a contextual subject grouping the set of concepts reachable from the parent. This can be used to avoid object-schizophrenia and to provide the functionality envisioned in subject-oriented programming.

5.9.2 Data Types, Content Types, and Qualified Content Types

At this point it is clear that a PDG can be used to model a useful subset of reality, including both concepts and the relationships between those concepts. However, the use of a PDG as the basis of a content-biased language (CBL) may still be somewhat mysterious. To help explain the relationship between the two, it is first necessary to introduce the notion of a content type, as distinguished from a data type.

In the context of a PDG, the data type for a particular concept is a subgraph of the PDG rooted at the associated concept node (called the distinguished node). The data type for car, with respect to the PDG shown in Figure 5-14, is illustrated in Figure 5-15. As shown in the diagram, the data type graph only contains concept nodes and unlabeled <ctx> nodes. Much like the type implied by a car class in an object-oriented type system, the data type for car includes attributes such as year, make, model, and VIN.

The problem with a data type is that it lacks context. To remedy this situation we introduce the notion of a content type. The content type for car, with respect to the PDG of Figure 5-14, is illustrated in Figure 5-16. Once again, the graph contains only concept nodes and unlabeled <ctx> nodes. However, the content type graph includes, in addition to the nodes of the data type graph, all nodes on a path from the root node of the PDG
down to the associated concept node. The inclusion of this path information provides the desired context for the concept.

![Diagram](image)

The distinction between a data type and a content type is primarily one of context. When discussing a content type, the location of the concept, within some specific PDG, is always made explicit. However, the true utility of the PDG model is best demonstrated by a third information type called the qualified content type. A simple qualified content type for car is illustrated in Figure 5-17. Whereas the content type shown in Figure 5-16 represented all cars relevant in the PBU for Car-Ownership, the qualified content type of Figure 5-17 places a restriction on the year of those cars. In particular, the qualified content type represents all cars of year 1999.
Content Type (Represents all cars in the model—if GUIDd then all cars in the world)
Car-Ownership/Car : Year, Make, Model, VIN

Figure 5-16. Content type

Qualified Content Type (All cars of year 1999)
Car-Ownership/Car : Year = ‘1999’, Make, Model, VIN

Figure 5-17. Qualified content type
A more complicated example of a qualified content type is shown in Figure 5-18, which represents all cars owned by “John Smith.” In this case, the restriction was placed on the name attribute of the person concept, rather than on the distinguished concept (car). Furthermore, the *owns* relationship between the restricted person concept and the car concept has been made explicit. It is important to realize that qualified content type graphs are still subgraphs of the original PDG with the additional possibility of restrictions on leaf nodes or ID attributes of particular concepts.

Figure 5-18. Qualified content type using a relationship node.

**5.9.3 PDGs in the Global System**

One or more PDGs are generally used, or associated with, a particular constituent system. Taken together, those PDGs define the subset of a language (CBL) spoken by that system. A constituent system designer may create new PDGs or may search a Global
Type Repository (GTR) to reuse existing PDGs. To avoid any confusion in terminology, recall that the PDG is our candidate model for an External Open Ontological Type System (EOOTS). Registering a PDG in the GTR is more accurately described as “sewing your wild EOOTS.”

There are two primary reasons for sewing your EOOTS. First, it motivates other constituent system designers to reuse your representation rather than create their own. Second, it enables one to map his EOOTS to other EOOTS, and thereby increase the probability for successful communication between his agents and other agents in the global system. Keep in mind that a single CBL may have several representations for the same concept (like synonyms in natural language). However, if these synonyms are mapped to each other then communication can still succeed. Of course, the preferred solution would be to have everyone speak using the same dialect, but this is unrealistic. Different market sectors may wish to represent the same content in different ways and support for mapping these dialects allows this to happen.

There are three primary mapping relationships listed below from weakest to strongest:

1. subjective comparability ($=_{SC}$): $c_1 =_{SC} c_2$ iff $c_1$ and $c_2$ share at least one direct descendant context node,

2. conceptual equivalence
   a. weak conceptual equivalence ($=_{CEW}$): $c_1 =_{CEW} c_2$ iff $c_1$ and $c_2$ have been mapped to each other and
   b. strong conceptual equivalence ($=_{CES}$): $c_1 =_{CES} c_2$ iff $c_1$ and $c_2$ have GUIDs that feed from the same GUID pool are in the same location in the same PDG, or $c_1 =_{CEW} c_2$ and $c_1$ and $c_2$ have ancestors related by strong conceptual equivalence, and

3. conceptual substitutability ($=_{CS}$): $c_1 =_{CS} c_2$ iff $c_1$ and $c_2$ have the same GUID or $c_1 =_{CEW} c_2$ and $c_1$ and $c_2$ have ancestors related by conceptual substitutability.
To help clarify the different mapping types, consider the car and truck concepts shown in Figure 5-14. Car and truck share the `<ctx s="vehicle">` node and, therefore, have subjective comparability. That is, one can compare the year of a car to the year of a truck. Subjective comparability is a special type of mapping that occurs between two different concepts in the same PDG. To accurately describe the other mapping forms, it is necessary to refer to a second PDG that models some of the same concepts illustrated in Figure 5-14. This new PDG, illustrated in Figure 5-19, represents a different dialect produced for a different problem domain. To keep the example simple, the PDG in Figure 5-19 was deliberately structured in a fashion similar to the PDG in Figure 5-14. However, the same concepts apply to any pair of PDG graphs.

![Home-Ownership PDG](image)

Figure 5-19. Home-ownership PDG

To describe weak conceptual equivalence, consider the person concept in both graphs. If, when the Home-Ownership PDG is registered, its person concept is mapped to the
person concept in the Car-Ownership PDG, then the two concepts have weak conceptual equivalence. To realize strong conceptual equivalence, we take into account the concept identification attributes of the two person concepts. If both concepts are decorated with a GUID (e.g., social security number), then they are said to have strong conceptual equivalence. Keep in mind that these concept identifiers must come from the same pool. That is, we cannot have strong conceptual equivalence if the person in Car-Ownership is identified with a social security number, and the person in Home-Ownership is identified with a tax identification number (unless an additional mapping existed between these two ID types). Finally, two instance of a concept have conceptual substitutability if they have conceptual equivalence and the concept identifiers are equivalent. That is, John and Jeff have strong conceptual equivalence (in that they are both people), but they do not have conceptual substitutability (since they are not the same person).

A more simple view of the different concept relationships is:

1. **Subjective comparability** - this relationship implies that the properties of two different nodes can be compared even though the nodes do not represent the same concept,

2. **Conceptual equivalence** - this relationship implies that two nodes represent the same concept, though they are not necessarily the same instance of that concept, and

3. **Conceptually substitutable** - this relationship implies that two nodes represent the same concept and are the same instance of that concept.

**5.9.4 PDG as a Service Description**

To make the global system truly useful, it must be possible for users not only to register information about themselves, but also to register intent. One possible solution to help a user convey his intention for registering some particular information is to use the context provided by the registration location. That is, if a user registers information
about his car in a matchmaker that specializes in car sales, then it can be assumed that the user intends to sell his car. This approach does not require a matchmaker for every possible service type. Instead, a single matchmaker could maintain many different rooms. A user would then register to enter into a particular room of the matchmaker and thereby convey the appropriate intent.

Recall from Section 5.6 the following three formulations of user information:

1. Fact – a fact is registered in the global system to advertise some property of its user. Examples include descriptions of the user’s possessions, personal profile information, and preferences,

2. Service Advertisement – a service advertisement is registered in the global system to advertise that the user can provide the described service on behalf of other agents. Examples include home-selling services, medical services, etc, and

3. Service Provision – a service provision is registered in the global system to advertise that the user is providing a certain service. Examples include selling a car or buying a car.

Using these formulations as a guide, the PDG model can be extended to include not only factual information but also service advertisement and service provision information. Consider the PDG illustrated in Figure 5-20. This extended Car-Ownership model includes a new relationship node called sell. By decorating relationship nodes, the sell node in particular, with an intent attribute, the same model can be used to convey three different statements. First, if <sell intent=\text{fact}>, which is the default, then the implication is that the referenced buyer sold the referenced item to the referenced seller. On the other hand, if an agent registers the PDG with the intent attribute set to advertise, it is assumed that the agent is advertising the fact that he will sell cars on behalf of other agents. Finally, if intent is set to provides the agent is stating that he would like to sell the referenced item.
While this chapter did cover a substantial amount of material, it is important to understand that all of the topics explored sought to solve the same underlying problem of enabling content transfer. The first step occurred when communication metadata was identified as a distinct and important component of any content transfer methodology. The required form of the communication metadata was in large part motivated by a realization that content-biased communication was sufficient for C2B applications. With a number of goals for content-biased communication made explicit, we then pursued a viable structure for the EOOTS. The EOOTS represent the basic building blocks of the GTR that ultimately defines the CBL used by a global system. The chapter concluded by examining the properties of one possible EOOTS model, the PDG, and how it could be used as a communication metadata representation.
CHAPTER 6
COMPONENTS OF A GLOBAL SYSTEM

In Chapter 3, Consumer-to-Business (C2B) systems were introduced as a new paradigm for consumer driven electronic commerce. In Chapter 4, the special requirements for a C2B system were clarified by introducing a new software quality measurement called stretchability. First, a set of properties called software specification axioms (SSAs) were defined as requirements gathering tools to be utilized in the software engineering process. Then, the concept of stretchability was refined by providing a formal interpretation based on the SSAs. One particularly important instantiation of an SSA, SSA-CommunicationIntent(ContentTransfer) formed the basis of Chapter 5, where the fundamental properties of a communication metadata language for supporting content transfer were explored. Throughout the discussion of Chapters 4 and Chapter 5, definitions were provided for many new ideas including: the Global System, Constituent Systems, the Global Type Repository (GTR), an External Open Ontological Type System (EOOTS), wild EOOTS, a Content-Biased Language (CBL), and finally a Perspective Domain Graph (PDG). In the present chapter, we present a high-level architecture for a Global System and its associated components.

6.1 Global System

In Figure 6-1, a global system is represented as the large circle in the center of the diagram. The constituent systems, represented as four darkened circles each intersecting with the boundaries of the large circle, can be seen to lie only partially within the confines of the global system. The choice of including exactly four constituent systems
in the diagram was arbitrary but is used to demonstrate the fact that many such systems will interact with the global system. As illustrated in the figure, the global system properly contains only those portions of the constituent systems designed as stretchable agents. To be sure, stretchable agents may also play a role in the design of other portions of a constituent system, and therefore some agents appear outside the scope of the global system in the figure.

Figure 6-1. Integrated representation of the major components in a global system

Taking a closer look at the constituent systems, four major component types are identified. The following list briefly describes each component, and the remainder of this chapter provides additional detail:
1. Salient Heap of Testimony (SHOT) – an aggregate of user related information stored in an arbitrary format, but accessible based on queries expressed using a content-biased language,

2. Pellets - self-contained packets of user information conveying a semantically relevant subset of data from a SHOT formatted as a set of statements in a content-biased language,

3. Agents – task oriented, possibly itinerant software modules, capable of communicating via a content-biased language, and

4. Wild EOOTS – domain specific communication metadata – which is possibly standardized by inclusion into the Global Type Repository through a process known as “Sewing one’s Wild EOOTS.”

Some of these component types also appear outside the context of a constituent system, but they arrive there only after being sent out from their parent constituent system. Once a component has crossed the boundary into the global system, it becomes part of the global system for the remainder of its stay. Some components will eventually return to their parent constituent system, while others will become indefinitely incorporated into the global system. The purpose and lifecycle of each component type is different, and we discuss each in turn.

The global system properly contains those portions of the constituent systems used for external communications, as well as currently active agents and pellets that have crossed the boundary. In addition, the global system is composed of a number of supportive components. These additional components, which collectively comprise the global framework, include the matchmakers, the replication facilitators, and the global type repository (GTR). The GTR, previously described in Chapter 5, is essentially a registered collection of wild EOOTS. Matchmakers perform many duties in the constituent system, including mediation, translation, and transaction coordination. The functionality of the matchmakers are discussed in greater detail throughout this chapter.
Finally, the replication facilitators ensure consistency of the information maintained by different pellets throughout a system. The functionality of the replication facilitator is not covered further, but its presence is included for completeness and to demonstrate the desired flexibility of the system.

Two additional, important details illustrated in Figure 6-1, concern the destination, or resting place, of injected agents and pellets. As a general rule, all agent-agent and agent-pellet interactions occur through an intermediary matchmaker. While an actual implementation may bend this rule for performance reasons, the logical implications remain intact. As a consequence, all agents shown within the global system are congregated around a matchmaker. Once within the global system, an agent is always in one of four states: traveling, negotiating for entry (into a matchmaker), registered (with a matchmaker), or involved in a transaction. In much the same way, a pellet is also always associated with some other component of the global framework. As illustrated in the diagram, a pellet may be held at a replication facilitator or may be registered within a matchmaker. As implied by the appearance of replication facilitators, it is not uncommon for a single pellet to appear in many places throughout the framework. Furthermore, there is no mandate requiring that the complete version of a pellet reside at every location. This coincides with the requirement for handling partial data identified in Chapter 5.

6.2 High-Level Component Interaction

Once a constituent system is developed, some of its itinerant modules will go online to perform their required tasks. This type of system architecture is somewhat different from standard client-server designs. Applications written to interact in the global system will generally be composed of a number of independent but cooperative agents. These agents
will break away from the home system to gather needed information and perform required tasks. This dynamic n-tiered view of computing diverges from the currently accepted notion of static tiers by permitting an application to move its modules wherever they may best be exploited. These modules may be data gatherers, negotiation experts, or GUI presentation objects. Regardless of their specific purpose, each module knows it belongs to a parent constituent system and works to achieve the goals of that system.

Figure 6-2 presents a high-level view of the creation and behavior of an agent-based constituent system. Each of the steps is described in greater detail throughout this chapter, and the following description should be referenced as often as needed. In step one, a constituent system agent is created using the (semi-automatic agent generation environment) SAGE. The agent goes online to retrieve some information. In the scenario illustrated, a GUI agent tags along with this initiating agent since there is a possibility that human interaction may be required to obtain the necessary information. In step two, the initiating agent arrives at a Matchmaker and strikes a deal based on the data in a user’s Pellet. However, the transaction requested by the initiating agent requires more data than the Pellet currently contains, so a GUI agent is sent to the user’s PDA. The user is notified of a new message on his PDA, and when he selects the message, he is greeted with the interface provided by the GUI-agent. In step three, the user allows the agent to access the additional data, and the agent returns to the Matchmaker. Having completed the transaction, the agents return to their home site. In step four, a persistence agent places the new information in a local database.
1. The agent application is created
2. The initiating agent strikes a deal with a Matchmaker
3. A GUI agent obtains additional data from a user
4. The completed transaction is logged by a data-centric agent

Figure 6-2. Activities of an agent-based constituent system

6.3 Components of the Global System

This section provides a general description of each major component of a global system including the SHOT, the pellets, the Matchmaker, and the SAGE. A discussion of the role played by agents appears in the following section in the context of a hypothetical transaction.

6.3.1 SHOT

As illustrated in Figure 6-3, the primary purpose of the Salient Heap of Testimony (SHOT) is to maintain a central repository of the user’s data. This data includes all content manually entered by the user or automatically updated as a result of the user’s transactions. Extracts can be distilled from the SHOT that still retain the semantics of the underlying information. Such extractions are Pellets and can be thought of as unique views into the original data. While a SHOT is considered a central store, it is possible to
create non-equivalent replicas with or without the ultimate goal of convergence. This allows user data to be stored at many locations and implies some form of distribution and/or replication mechanism. It is important to note throughout the discussion that the data stored in the SHOT can be representative of a person, a business, or any other entity. In fact, the design of the entire architecture is not limited to utilization for only personal data. However, since our present aim is to support C2B, the conversation is generally geared in that direction.

![Figure 6-3. SHOT, a central repository for user data](image)

**6.3.2 Pellets**

Pellets, as described above and illustrated in Figure 6-4, are subsets of a user’s data that retain the semantic context from the original location. Companies, Matchmakers, or other entities may choose to store a user’s personal data pellets in their own databases.
As the data changes throughout the course of everyday interactions, all of these parties may be interested in receiving notifications regarding any modifications. These notifications imply the existence of a registration mechanism, as well as an automated update procedure. Consider the likely scenario of a set of non-disjoint Pellets scattered in various locations across the Internet. This provides redundancy and improves efficiency, but it also motivates the need for novel approaches to replication and distribution.

![Diagram of Pellets, many views on the data](image)

While storage of Pellets is extremely important, their primary use is really the communication of content between the user and any interested parties. Users will place security restrictions on the various aspects of their content. Then, when new requests for information are received, the restrictions will dictate the security level at which requesters shall be entitled to perform the access. In addition, mobile users will likely carry a number of very important Pellets on their PDAs. These Pellets will be used during business transactions, for remembering names during social engagements, or in emergency situations. In all cases, the data in the Pellets are likely to be read as well as
written. The reconciliation of the changes back at one or more central repositories poses a number of unique challenges.

6.3.3 Matchmaker

Once the Pellets have left their constituent system they may be unwound and stored in a database, transferred as is to another user’s constituent system, or wind up in a Matchmaker. As shown in Figure 6-5, a Matchmaker is an independent module that provides a barrier preventing unauthorized access of Pellet data by curious agents. In most cases, a Matchmaker will act as a mediator assisting negotiations in some specific business domain. Users will send their Pellets to a Matchmaker because they are interested in products associated with that particular domain, and businesses will send their agents to a Matchmaker because they sell products related to the domain. However, there is no guarantee that the Matchmaker will accept either the Pellets or the agents. Much like the cell of a biological organism, a Matchmaker exposes a semi-permeable membrane to the outside world. This membrane implements acceptance criteria for Pellet data and agents and keeps the inside of the Matchmaker clean and efficient. Other duties of the Matchmaker include maintenance of pellet data at replicated sites, assistance in interfacing between agents and pellets, assistance in agent / enhanced pellet negotiations, implementation of a security protocol blocking the activities of destructive agents, and proactive discovery of worthwhile agents or Pellets.
6.3.4 Semi Automatic Agent Generation Environment

An optional component of the architecture is the Semi-Automatic Agent Generation Environment (SAGE)\(^1\). The SAGE, illustrated in Figure 6-6, is a user-friendly development tool that simplifies creation of transaction-aware agents. Agents created using the SAGE are guaranteed to interoperate with compliant Matchmakers and to comprehend the data maintained in compliant Pellets. Furthermore, these agents will have inherent support for mobility and will have the capacity to communicate with other agents using a content-biased language. Though the SAGE is not a mandatory element of the C2B architecture, its existence would surely accelerate the acceptance of the new technologies. While one of our research objectives is to standardize infrastructure to

---

\(^{1}\) The SAGE introduced here should not be confused with the Scalable Adapter GEnerator (SAGE) described by IBM. That SAGE is used to simplify construction of inter-component adapters.
simplify the creation of an agent-based application, a significant amount of effort will still likely be required to produce a skeleton implementation. The SAGE exists to automate that repetitive effort and allow the developer to spend more time working on the details of his specific problem domain. Given this description, it is fair to say that the SAGE plays a role similar to that of an integrated development environment (IDE) for building constituent systems (particularly the stretchable portions). And, like an IDE, the SAGE is not considered a component of the systems it helps to build (hence its absence from Figure 6-1).

Figure 6-6. SAGE

6.3.5 Useful Perspective on the Global System

This section provided a high-level view of the global system components. The illustration in Figure 6-7 provides a useful perspective on the general flow of information
throughout the system. This global view of the system shows how the SHOT, SAGE, Pellets, and Matchmakers interact across the Internet to provide a robust means for the exchange and processing of user content. The details of these interactions, especially with respect to transactions, the EOOTS, and the GTR are discussed in section 6.4 and Chapter 7.

Figure 6-7. High-level C2B system architecture

6.4 Global System Transaction

To provide a running example for the discussion of this section, we refer the reader to the illustration in Figure 6-8. The diagram shows a static snapshot of a multi-agent interaction. To avoid the preconceptions that arise when considering real-life transactions, the example is based on the sale of a generic widget object. When a more concrete example is required, the widget can be instantiated as a home, an automobile, or any other appropriate concept.
Referring to Figure 6-8, the agent, $A_{\text{buyer}}$, wishes to purchase a widget, $W$, from a
widget broker, $A_{\text{broker}}$. The widget broker is selling the widget on behalf of the owner,
$A_{\text{seller}}$. Furthermore, the buyer requires insurance on the widget and will choose between
$A_{\text{insurer1}}$ and $A_{\text{insurer2}}$. In addition to the matchmaker, $M$, there may also be a replication
facilitator (not shown in the diagram), that handles issues, such as replication of the
buyer’s insurance information. The following outline of the actions required to initiate a
multi-agent transaction corresponds to the numbers shown in Figure 6-8 and provides an
excellent reference point for the remaining discussion.
Building Up to a Multi-Agent Transaction

1. The seller’s constituent system injects an agent and a pellet into the global system. The agent/pellet combination negotiate for entry into the matchmaker, and assuming they are accepted …

2. The pellet, \( P_W \), is registered with the matchmaker, and …

3. The agent, \( A_{seller} \) is registered with the matchmaker.

4. A broker’s constituent system sends out a broker agent to help the seller with its business. The agent negotiates for entry into the matchmaker and, assuming it is accepted, …

5. The broker agent, \( A_{broker} \) is registered with the matchmaker.

6. A buyer’s constituent system sends out an agent to investigate possible opportunities. The agent negotiates for entry into the matchmaker and, assuming it is accepted, …

7. The buyer agent, \( A_{buyer} \), investigates the data available in the various pellets registered with the matchmaker and finds one of interest. The buyer agent then requests that the matchmaker initiates a transaction. The matchmaker initiates the transaction and notifies all agents registered against the pellet (in this case, \( A_{seller} \) and \( A_{broker} \)).

8. While the transaction is under way, an insurance agent \( A_{insurer1} \), negotiates for entry into the matchmaker, and assuming it is accepted, …

9. \( A_{insurer1} \) attempts to take part in the ongoing transaction.

10. Also, while the transaction is under way, another insurance agent \( A_{insurer2} \), negotiates for entry into the matchmaker, and assuming it is accepted …

11. \( A_{insurer2} \) attempts to take part in the ongoing transaction.

At this point, the outline has reached the state illustrated in Figure 6-8. The transaction continues until a deal is reached, or the parties choose to terminate their negotiation. The details on how the transaction participants communicate is discussed in Chapter 7, but as expected, it is based on the use of a content-biased language with statements structured as perspective domain graphs. The remainder of this chapter
examines each of the steps outlined above in more detail, and in so doing, clarifies the 
role of the matchmaker and the different agent types.

6.4.1 Detailing the Role of the Matchmaker

Glancing again at the illustration in Figure 6-8, the reader is reminded of the central 
role of the matchmaker, M. As described in section 6.3.3, a matchmaker is a mediator 
that assists negotiations in some business domains. A matchmaker exposes a semi-
permeable membrane that selectively filters incoming agents and pellets. In the present 
example, the matchmaker, M, supports business transactions related to the purchase and 
sale of widgets.

The first action involved in initiating the purchase transaction is the registration of the 
widget, W, by the seller. Note that in the introduction to section 6.4, we seem to have 
endowed the agents with the human characteristics of desire and intention. That is, we 
stated that $A_{\text{seller}}$ wished to sell its widget, W. In reality, our model does not depend on 
such notions, though they are often convenient for exposition. To make this distinction 
more concrete, we clearly distinguish between the entity, the seller, and the entity’s 
agent, $A_{\text{seller}}$. The entity is most likely a business or a human, but there are no technical 
impediments to realizing the entity as an “intelligent” agent. In the latter case, the 
distinction between the entity and its agent is not strictly necessary, though there is likely 
to be a containment relationship between the two. We must leave an ownership 
relationship discussion to philosophers and lawyers.

Registration of the widget, W, with the matchmaker, M, entails transfer of a pellet. It 
is assumed that the seller maintains a salient heap of testimony (SHOT), and contained 
within that SHOT, is information pertaining to the aforementioned widget. The 
extraction of the widget information from the SHOT yields a pellet, $P_W$, containing a
description of the widget. The seller sends the pellet, \( P_w \), to the matchmaker, \( M \), requesting that it be accepted. When the pellet reaches the matchmaker, it first encounters the semi-permeable membrane and either continues through or is rejected. Assuming the pellet enters the matchmaker, it is registered, and the widget is now available for sale. At this point, the matchmaker has explicitly accepted the pellet and implicitly accepted the responsibility for overseeing access to the pellet’s information. Furthermore, the matchmaker has agreed to offer itself as a transaction-aware container for any agents in any transactions relating to the pellet.

The previous paragraph highlights two important points. First, the matchmaker is not simply a static repository for pellet data. Not only does it selectively accept or reject pellets, it also enforces security measures to restrict unauthorized access to portions of the pellet information. Second, the matchmaker assumes all the necessary functionality of a distributed transaction coordinator including possible interactions with a replication facilitator. Due to the varying nature of the data maintained within pellets, a matchmaker may be required to implement many different transaction models. Furthermore, the matchmaker may also be called upon to act as a non-coordinating participant of an existing transaction.

**6.4.2 Transaction Participants**

Throughout this chapter, agents have been described as the primary actors, or transaction participants, in the system. Given the importance of these agents, it is now fitting to discuss them, where they come from, and how they involve themselves in transactions. We have already previewed the participants in the introduction to section 6.4 when we described the purpose of \( A_{seller} \), \( A_{buyer} \), \( A_{broker} \), \( A_{insurer1} \), and \( A_{insurer2} \). Each of
these agents will play a role in the transaction, though their “desires” and “intentions” may be entirely independent.

As we have seen, the seller registered the pellet, $P_w$, with the matchmaker. In broad terms, the pellet represents the shared resource (or one of the shared resources) in the impending purchase transaction. Since the seller may have particular requirements surrounding the sale of his widget, he may also choose to register an agent, $A_{\text{seller}}$, to look after the resource. Once engaged in a transaction, $A_{\text{seller}}$ may act to constrain the results in favor of the seller. In addition, $A_{\text{seller}}$ may report results, contact the seller during the transaction, or perform any number of other seller-related tasks. At the present time, the important point to remember is that $A_{\text{seller}}$ is registered by the seller any time after registration of $P_w$ and is explicitly associated with the pellet $P_w$. The importance of the latter point will become clear later.

6.4.2.1 Broker agent (not a matchmaker)

The broker agent, $A_{\text{broker}}$, requires some initial clarification. It is important to distinguish between the matchmaker, $M$, and the broker. In our model of the global framework, a matchmaker plays a role similar to what others have entitled brokering. In fact, the terms, *facilitator*, *matchmaker*, and *broker* are often used interchangeably in the literature. For the sake of clarity, we have chosen to use the term *matchmaker*, and only matchmaker, to refer to the distinguished role described in section 6.3.3. The broker agent, $A_{\text{broker}}$, has no special status as part of the framework and is simply another participant in the purchase transaction. Thus, it should be clear that not all transactions require a broker, whereas they do all require a matchmaker. To provide a concrete example, consider that the broker will play the same role in our model as a real-estate agent does in the physical world. Given this fact, it is important to consider how $A_{\text{broker}}$
comes to be involved in the transaction, or more generally, how the broker ingratiates himself to the seller.

6.4.2.2 Establishing agent relationships

If we continue with the analogy to home sales, it becomes evident that there is no single model for initiating the association between a seller and a broker, nor is there any restriction on the number of brokers that may be involved in a transaction. For example, a seller may first attempt to advertise his home without any outside assistance. This is commonly referred to as for sale by owner. The analogous case in our model would be a transaction involving no brokering agents. Assuming there is a broker, the interposition commonly occurs through one of two primary alternatives. Either, the seller loses patience and seeks outside assistance, or the broker contacts the seller and convinces him to sign (accept a contract with the broker). To maximize its potential utility, the global system should support both alternatives.

In the first option, the seller seeks out the broker. This activity could be accomplished outside of the global system through direct human contact. If that is the case, then when the seller registers its agent, \( A_{\text{sellers}} \), with the matchmaker, it also submits a reference to the broker’s agent, \( A_{\text{broker}} \), presumably provided by the broker during the human interaction. The matchmaker may be implemented to proactively seek the broker’s agent based on the reference information, or it may simply accept the broker agent’s request for registration when, and if, it arrives. An alternative to this scenario is the creation of a separate transaction used to establish a seller-broker relationship prior to enabling the intended buy-sell transaction. In this case, the seller registers the widget pellet, \( P_W \), and also submits its intention to engage a satisfactory broker.
Returning to the issue of establishing a seller-broker relationship, we now consider the alternative in which the broker actively seeks out the seller. This case is very similar to the previously mentioned option where the seller expressed a desire to locate a satisfactory broker. However, whereas that option entailed explicit direction from the seller, this alternative simply assumes the seller has not requested otherwise. That is, the seller has not informed the matchmaker that it desires all requests from brokers to be blocked. Given these parameters, upon receipt of a broker request, the matchmaker may notify the seller and obtain additional instructions, possibly resulting in the initiation of a transaction to examine the feasibility of establishing a relationship. In this example, it is assumed that the broker agent has access to some portion of the widget pellet information, which it uses to determine whether it is interested in brokering, and if so, under what conditions.

Other alternatives leading to the involvement of broker agents are also possible. For instance, the buyer may establish a relationship with a broker, or perhaps an insurance agent. Furthermore, as the complexity of the possible transactions increases, it may be impossible to predetermine the ultimate number of brokers, or participants in general. Thus, a goal of the global system is to enable dynamic interposition and withdrawal without affecting the integrity of the transaction. This may require the introduction of sub-transactions and portends the need for nested-transactions and other advanced transaction models.

The previous discussion clearly distinguishes between two types of participants. The first is an enlisted participant that registers with a matchmaker and requests notification when accesses occur against a given resource. That resource is either a previously
registered pellet, or a pellet that the enlisted participant carries (and then registers). In either case, the enlisted participant and the pellet are both associated with the same constituent system. They were either both created by the same constituent system, or the enlisted participant agent was acquired and conditioned by the same constituent system that created the pellet. The seller agent, \( A_{\text{seller}} \), is an example of an enlisted participant. As described above, after the pellet, \( P_W \), is registered at the matchmaker, the agent, \( A_{\text{seller}} \), is registered and requests association with the pellet. Following that, the seller agent waits for external activity against its resource.

**6.4.2.3 Opportunistic participants**

The second participant type, an opportunistic participant, can be further decomposed into two related subtypes. A static opportunistic participant registers against a particular pellet at the behest of an enlisted participant. A good example is the broker agent, \( A_{\text{broker}} \). The seller agent, \( A_{\text{seller}} \), has chosen to enlist the services of the broker agent and has requested (permitted) that it register against the pellet, \( P_W \). Now, when a transaction is requested against the pellet, both \( A_{\text{seller}} \) and \( A_{\text{broker}} \) will be immediately involved. In fact, \( A_{\text{broker}} \) may do all of the work, and \( A_{\text{seller}} \) can simply accept or decline the final proposal. In contrast, a dynamic opportunistic participant joins a transaction after it has already begun, usually at the invitation of some other participant. For instance, a broker agent could become a dynamic opportunistic participant if the chosen insurance agent requested its inclusion. However, better examples of dynamic opportunistic participants are the insurance agents themselves. Before describing the roles of the insurance agents, however, it is useful to examine the buyer agent. Also, it is important to remember that in our example, the broker agent plays the role of a static opportunistic participant – once found, it registers itself against the pellet and “waits” for a buyer.
6.4.2.4 Initiating participants

The buyer agent motivates extension of the participant classification system through introduction of a third participant type. An initiating participant is neither enlisted nor opportunistic; its arrival is unpredictable, although it always occurs immediately prior to the start of the transaction. The latter point is predicated on the fact that the transaction begins at the request of the initiating participant. Consider the state of our running example just before the buyer agent, \(A_{\text{buyer}}\), arrives (disregarding the eventual involvement of the insurance agents). The matchmaker has accepted registration of the pellet, \(P_W\), and the agents \(A_{\text{seller}}\) and \(A_{\text{broker}}\). At this point, no activity is taking place as the seller and broker are awaiting the arrival of an interested buyer. Even if the broker is actively seeking buyers, the purchase transaction will not begin until an interested individual is located. Thus it is the appearance of the buyer that initiates the transaction.

Once again, an important distinction must be made. When discussing participant roles, it is vital to provide the context of a particular transaction. In the discussion above, we have focused on the purchase transaction involving \(A_{\text{seller}}, A_{\text{buyer}}, \) and \(A_{\text{broker}}\). However, consider the activities that occur if the broker agent is actively searching for buyers on behalf of the seller. In this case, it is possible that transactions involving the buyer and broker will occur without the participation of the seller. Relative to these broker-buyer transactions, the broker may be considered an initiating participant. Thus, the participant type of a given agent is relative to a specified transaction and is not a static property of the agent itself.

Returning to our discussion of the insurance agents, \(A_{\text{insurer1}}\) and \(A_{\text{insurer2}}\), we can now examine the role of a dynamic opportunistic participant in an existing transaction. Once an interested buyer has been located, or has discovered the opportunity on his own, the
buyer makes clear his intent to initiate a transaction. Or, in terms of our example global system, \( A_{\text{buyer}} \) notifies the matchmaker, \( M \), of its desire to purchase the widget described by \( P_W \). As discussed above, \( P_W \) has a number of agents associated with it; namely, \( A_{\text{seller}} \) and \( A_{\text{broker}} \). In response to the buyer’s solicitation, the matchmaker initiates a transaction and invokes \( A_{\text{seller}}, A_{\text{broker}}, \) and \( A_{\text{buyer}} \) as participants. During the course of the transaction, it may occur that \( A_{\text{buyer}} \) insists on the involvement of an insurance agent. A reasonable explanation for this is that the buyer cannot complete the purchase if he is unable to secure a sufficient insurance policy within his budgetary restrictions. Temporarily, we will leave unanswered the question of how the insurance agents are located and simply submit the fact that two viable agents, \( A_{\text{insurer1}} \) and \( A_{\text{insurer2}} \) are discovered. These two insurance agents are competitors, so only one will ultimately be selected. However, initially, both insurance agents will enter the transaction and act as participants. As described, neither of the insurance agents was associated with the pellet or any other aspect of the transaction prior to their invitation. Furthermore, one of the insurance agents will leave the transaction prior to its completion. It is these behaviors that lead us to call these agents dynamic opportunistic participants.

6.5 Clearly Distinguishing the Participant Types

At this time it is helpful to examine the lifecycle of the different roles agents play in the system. Three primary participant types were identified above, namely enlisted participants, opportunistic (static and dynamic) participants, and initiating participants. While the lifecycles involved with each of these participant types are very similar, there are some important distinctions. Formalizing the different lifecycles will help clarify the distinctions and will also provide a set of functional requirements for
States of an Enlisted Participant’s Lifecycle:

1. [Passive] Creation of the agent that will play the enlisted participant role.
2. [Passive/Active] Acquisition of the agent.
3. [Passive/Active] Conditioning of the agent for the intended purpose.
4. [Passive/Active] Injection of agent into global system from constituent system.
5. [Active] Traveling within the global system.
7. [Active] Registration of any pellets the carried by the agent.
8. [Active] Registration with matchmaker against a particular pellet.
10. [Active] Participation of the agent in a transaction started by an initiating participant.
11. [Active] Deregistration with the matchmaker.
13. [Active] Traveling within the global system.
14. [Passive/Active] Egression of agent from global system into constituent system.
15. [Passive/Active] Extraction of results from agent.

The States of a Static Opportunistic Agent’s Lifecycle are:

1. [Passive] Creation of the agent that will play the static opportunistic participant role.
2. [Passive/Active] Acquisition of the agent.
3. [Passive/Active] Conditioning of the agent for the intended purpose.
4. [Passive/Active] Injection of agent into global system from constituent system.
5. [Active] Traveling within the global system.
6. [Active] Negotiation between agent and semi-permeable matchmaker (M1) membrane for entry.
7. [Active] Registration of any pellets carried by the agent.
8. [Active] Registration with matchmaker (M1) against a particular pellet.
9. [Active] Registration with matchmaker (M1) against a particular pellet/service template.
10. [Active] Accept invitation to assist enlisted participant in its transaction.
11. [Active] Deregistration with the matchmaker (M1).
12. [Active] Negotiation between agent and semi-permeable matchmaker (M1) membrane for exit.
13. [Active] Traveling within the global system.
14. [Active] Negotiation between agent and semi-permeable matchmaker (M2) membrane for entry – with assistance of benefactor enlisted participant.
15. [Active] Registration with matchmaker (M2) against enlisted participant’s associated pellet.
16. [Active] Participation of the agent in a transaction with the enlisted participant started by an initiating participant.
17. [Active] Deregistration with the matchmaker (M2).
18. [Active] Negotiation between agent and semi-permeable matchmaker (M2) membrane for exit.
19. [Active] Traveling within the global system.
20. [Passive/Active] Egression of agent from global system into constituent system.
22. [Passive/Active] Liberation of the agent.
The States of a Dynamic Opportunistic Agent’s Lifecycle are:

1. [Passive] Creation of the agent that will play the dynamic opportunistic participant role.
2. [Passive/Active] Acquisition of the agent.
3. [Passive/Active] Conditioning of the agent for the intended purpose.
4. [Passive/Active] Injection of agent into global system from constituent system.
5. [Active] Traveling within the global system.
6. [Active] Negotiation between agent and semi-permeable matchmaker (M1) membrane for entry.
7. [Active] Registration of any pellets carried by the agent.
8. [Active] Registration with matchmaker (M1) against a particular pellet.
9. [Active] Registration with matchmaker (M1) against a particular pellet/service template.
10. [Active] Accept invitation to participate in a transaction.
11. [Active] Deregistration with the matchmaker (M1).
12. [Active] Negotiation between agent and semi-permeable matchmaker (M1) membrane for exit.
13. [Active] Traveling within the global system.
14. [Active] Negotiation between agent and semi-permeable matchmaker (M2) membrane for entry – with assistance of host participant.
15. [Active] Participation of the agent in the currently running transaction.
16. [Active] Negotiation between agent and semi-permeable matchmaker (M2) membrane for exit.
17. [Active] Traveling within the global system.
18. [Passive/Active] Egression of agent from global system into constituent system.
20. [Passive/Active] Liberation of the agent.
The States of an Initiating Agent’s Lifecycle are:

1. [Passive] Creation of the agent that will play the dynamic opportunistic participant role.
2. [Passive/Active] Acquisition of the agent.
3. [Passive/Active] Conditioning of the agent for the intended purpose.
4. [Passive/Active] Injection of agent into global system from constituent system.
5. [Active] Traveling within the global system.
6. [Active] Negotiation between agent and semi-permeable matchmaker (M1) membrane for entry.
7. [Active] Exploration of pellets that have been registered with matchmaker (M1) by enlisted participants.
8. [Active] Initiation of transaction against a particular pellet.
10. [Active] Traveling within the global system.
11. [Passive/Active] Egression of agent from global system into constituent system.
12. [Passive/Active] Extraction of results from agent.

6.6 Interpreting the Agent Lifecycles

At first glance, it may seem that we have eliminated the possibility of proactive opportunistic participants. For example, agents, such as the brokers described above, appear to be unable to actively seek opportunities. In the lifecycles of both static and dynamic opportunistic participants, the agents seem to wait for an invitation into a transaction. This is true, but we must recall that participant type is tied to a particular transaction. Hence, if a broker agent wants to search for opportunities, it may do so, but
only while assuming the role of an initiating participant. The transaction it initiates is one in which the customer decides whether or not to use the broker’s services, and under what conditions. Then, in the customer’s transaction, the broker plays the role of an opportunistic participant. This observation raises two interesting questions. First, do we allow presently running transactions to be viewed and interrupted by uninvited participants? And, second, is the same agent that proactively seeks opportunities also involved in handling the new transactions?

6.6.1 Uninvited Participants

The answer to the first question is really a matter of preference. For some transactions, security and privacy may be of utmost importance, while for other dealings the potential benefits resulting from global visibility may be the primary consideration. Therefore, the answer to the question is that in-progress transactions may be externally perceptible, but only if all participants agree. If any participant declines, then the remaining participants can attempt to replace the dissenter, or they can simply acquiesce to its demands. Dynamic interposition of uninvited participants into an executing transaction should not cause any insurmountable problems since the initial participants need simply ignore the solicitations. If the interruptions become too frequent, or if the transaction has reached a point where interruptions are no longer appropriate, then the participants can inform the matchmaker to block any further attempts.

6.6.2 Ancillary Agents

The answer to the second question provides an excellent segue into the next topic, ancillary agents. An ancillary agent performs non-transaction related tasks either in support of an existing transaction or to determine whether it is appropriate to initiate a transaction. To explain further, we turn to a real world example. In a stockbroker's
office, it is quite common for a novice broker to spend much of his day making unsolicited calls, or cold calls, to search out potential customers. The majority of these cold-calls will not result in worthwhile clients for the firm. However, by executing a large number of such calls, even with the smallest probability of success, a sufficient number of valuable leads can be uncovered. Once a number of leads are compiled, they are passed on to the trained brokers who attempt to initiate a transaction and hopefully close a deal. This same model can be applied to separate the responsibilities of different agents in the global system. For example, a lightweight cold-call agent can travel the global system searching for potential opportunities. This search may entail nothing more than examination of registered pellet data. If the data matches some established pattern, then the cold-call agent can notify the broker agent regarding a possible opportunity. The broker agent can then start a transaction where it plays the role of an initiating participant, and the pellet owner plays the role of the enlisted participant. If the pellet owner agrees to accept the services of the broker agent, then the broker agent will either register against the pellet or simply wait for an invitation into any transactions involving the pellet owner. In the former case, the broker becomes a static opportunistic participant, and in the latter it becomes a dynamic opportunistic participant. There is certainly no reason why the broker agent cannot cold-call for itself, but it probably makes better sense to disseminate a lightweight agent to seek opportunities. Since the name cold-call agent is somewhat limiting, we will refer to these agents as solicitation agents.

There are certainly many examples of ancillary agents, but for the present time, we will only highlight two additional instances. A presentation agent is used to display a graphical user interface (GUI) to the human participant in a transaction. During the
course of a transaction, a situation may arise where additional data is required but is not available from any of the participant agents. When this occurs, a presentation agent can be dispatched for communication with the human user. If the user has a personal digital assistant (PDA), the presentation agent can display a GUI on the PDA. The user can then answer the required questions and send the data back into the transaction. Through this mechanism, a transaction that is missing some needed data does not have to fail. Rather, it can simply persist in its current state, send out a presentation agent, and then resurrect itself when the required data arrives. Enabling this technique requires support from the matchmaker, namely in assisting in the persistence and resurrection of the current transaction.

The final type of ancillary agent we discuss is called a persistence agent. As suggested by its name, a persistence agent helps other agents store their data into some reliable medium. If, during the activities of a transaction, a participant wants to save certain important data, it can simply request the services of a persistence agent and assume that the data will reach its intended destination. This is useful in situations where the data being saved is not vital in relation to the transaction; that is, the transaction fails or succeeds independent of whether the persistence agent is successful in its task. Real world applications of this functionality may include auditing, logging, and debugging.

6.7 Summary

In this chapter, we explored a hypothetical architecture for a global system and its associated constituent systems. Components, such as SHOTs, Pellets, Matchmakers, Agents, and SAGEs were discussed. Additional details of agents and the matchmaker were examined in the context of a global system transaction. The requirements of the transaction motivated identification of four agent types including enlisted participants,
initiating participants, static opportunistic participants, and dynamic opportunistic participants. The lifecycles of these different participant types were also discussed, and a number of additional ancillary agent types were explored.
CHAPTER 7
DETAILING A C2B SYSTEM DESIGN

Current trends in distributed system design revolve around the notion of web services. These web accessible software modules leverage a number of new technologies including XML, SOAP, WSDL, and UDDI, as well as a significant amount of underlying infrastructure provided by runtime environments and application servers. While many of these technologies are relative newcomers, the basic software design principles applied in their utilization remain much the same. In general, the interface is separated from implementation, and some type of naming (or directory) service is employed to help locate applicable implementations at runtime. Though reflection and other forms of dynamic interface discovery are available, the primary means of encoding functionality is still based on a client’s design-time adherence to a well-known interface contract. Furthermore, the chief mechanism for obtaining that functionality at run-time is still a fixed-signature, one-way method call. Of course, the method no longer needs to reside in the process space of the client, and the actual implementation of the method need not be discovered until run-time, but aside from that, the protocols engaged when using today’s cutting edge technology differ little from those of the last ten years. The two advantages mentioned previously have also been utilized for quite some time in the form of DCOM and EJB.

To be sure, it is not our intention to diminish these recent technological advances. In fact, all of the new technologies and infrastructure provide a solid foundation on which to build our proposed C2B system. However, to compliment this new suite of tools and
technologies, we require a fundamental shift in the currently accepted software
development methodologies. This paradigm shift is based on the use of a content-biased
language to enable an open dialog between software agents. In this chapter, we review
the new technologies mentioned above and then demonstrate how they can be applied to
the design of a C2B system.

7.1 Current Technologies

This section reviews a number of new web-based software technologies including
SOAP, WSDL, and UDDI. At present, many of these standards have not been finalized.
However, robust implementations already exist, and the general direction and scope of
the technologies seem well defined. This section is not a tutorial and only provides a
basic overview of each topic.

7.1.1 Simple Object Access Protocol

The Simple Object Access Protocol (SOAP) was originally designed to support XML-
based remote procedure calls (RPCs) over the HyperText Transfer Protocol (HTTP). At
present, SOAP has been recast as a general-purpose XML-based messaging protocol for
use over any transport protocol (e.g., HTTP, SMTP, etc.), and the object/RPC suggestive
acronym has been dropped. An alternative acronym, Service-Oriented Architecture
Protocol, was proposed, but at the time of this writing was not accepted. Distilled to its
essence, SOAP simply provides a well-known format for packaging messages in what is
called an envelope. These envelopes are sent from an initial SOAP sender, through one
or more SOAP intermediaries, to the ultimate SOAP receiver. When the message reaches
the ultimate SOAP receiver, its contents are interpreted resulting in a method call (if
using SOAP for RPC) or in some other behavior.
The basic format of a SOAP envelope is illustrated in Figure 7-1. As shown in the figure, the SOAP envelope contains a header portion and a body portion. The header portion is composed of SOAP header blocks, which are generally used to convey information orthogonal to the primary intention of the SOAP message. For example, SOAP header blocks may encode details related to transaction identification, authentication, or information regarding the targeting of particular SOAP nodes to handle processing of the message. The SOAP standard does not concern itself with formalizing possible uses for header blocks, but rather provides a standard extension mechanism for application specific enhancements.

![SOAP Message Diagram]

The body portion of the SOAP message contains a set of body blocks used to convey the primary intention of the message. The content of the body blocks generally takes one of two forms, document or RPC. When RPC is chosen, the body blocks encode the necessary information to perform a method call. Such information usually includes the method name and the values to be passed for each of the [in] and [in/out] parameters. A
document-based SOAP message, on the other hand, is not tied directly to a method call. Instead, the message conveys an XML document that can be interpreted by the ultimate SOAP receiver. How the receiver reacts to the message is totally dependent on the particular application. An example of using SOAP to perform an RPC using the HTTP POST follows.

Method Signature

public Boolean queryCars(string make, string model, int year, out queryCarStruct[] cs)

SOAP Request

POST /ws/CarWS.asmx HTTP/1.1
Accept: */*
Accept-Language: en-us
soapaction: http://C2B/ExampleCarItf/queryCars
Content-Type: text/xml
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible;MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)
Host: localhost:8080
Content-Length: 332
Connection: Keep-Alive
Cache-Control: no-cache

<soap:Envelope xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
 xmlns:soap = "http://schemas.xmlsoap.org/soap/envelope/"
><soap:Body>
 <queryCars xmlns="http://C2B/ExampleCarItf">  
 <make>Chevrolet</make>
 <model>Corvette</model>
 <year>2001</year>
 </queryCars>
</soap:Body>
</soap:Envelope>

SOAP Response

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Thu, 21 March 2002 03:46:28 GMT
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8
Content-Length: 708
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <soap:Body>
  <queryCarsResponse xmlns="http://C2B/ExampleCarItf">
   <queryCarsResult>true</queryCarsResult>
   <cs>
    <queryCarStruct>
     <vin>A12345BCD123456VIN123456</vin>
     <price>14999.99</price>
     <mileage>1500</mileage>
    </queryCarStruct>
    <queryCarStruct>
     <vin>A23456BCD123456VIN123456</vin>
     <price>24999.99</price>
     <mileage>2500</mileage>
    </queryCarStruct>
    <queryCarStruct>
     <vin>A34567BCD123456VIN123456</vin>
     <price>34999.99</price>
     <mileage>3500</mileage>
    </queryCarStruct>
   </cs>
  </queryCarsResponse>
 </soap:Body>
</soap:Envelope>

7.1.2 Web Services Description Language

The Web Services Description Language (WSDL) is the interface description
language (IDL) used for describing web services. WSDL is used to define the interfaces
exposed by a web service, including all methods and their signatures as well as the access
protocols and locations for calling those methods. The general form of a WSDL
document appears in Figure 7-2. Note that the sections decorated with an asterisk (*)
may repeat any number of times.

The <types> section of the WSDL document is used to define any special data types
that may be required to access the web services. When dealing with RPC calls, these
descriptions often provide type information for each of the method parameters. Instead
of providing the type definitions directly inside of the WSDL document, one may also
use an <import> element to include an external XML-Schema document. This latter
alternative may be used to increase modularity and also to motivate reuse of the type definitions across many different services.

### Figure 7-2. WSDL document structure

There may be any number of `<message>` elements included in the WSDL document. Each `<message>` element may contain any number of child `<part>` elements used to define the format of the data being passed in the message. The format specifications may appear directly as XML-Schema or may reference the types defined in the `<types>` section of the WSDL file. For a request/reply scenario, there would normally be two messages defined, one for the request and one for the reply. In the context of an RPC call, the request message would contain parts for each `[in]` and `[in/out]` parameter, while
the response would contain parts for the optional return value as well as an [in/out] and [out] parameters.

Just as a <message> element may be compared to a method declaration, a <portType> element can be compared to an interface definition. That is, a <portType> section defines the set of operations supported by a particular web service. Each operation is defined using an <operation> element that in turn references either one or two <message> elements. For a one-way operation or notification, where the end-point either only receives or sends a message, there will only be one <message> referenced in the operation definition. For a bi-directional (e.g., request-response or solicit-response) message, such as a function call, the corresponding operation will reference both a request message and a response message.

The <portType> elements close what is frequently referred to as the interface definition portion of the WSDL document. However, as a UDDI tModel (discussed in the following section) may only refer to a WSDL document containing at least one <binding> section, it is not uncommon to include bindings as part of the service interface description. Since the Web Service interface defined by a <portType> may be accessed using any available transport protocol, it is necessary to provide a binding section for each protocol/interface pair. Thus, if a particular interface were accessible through both HTTP/GET and SOAP then two bindings would be required. The first binding would detail information concerning how the messages would be structured within the query string, while the second binding would specify items such as the SOAP action.

The final section of a WSDL document comprises what is frequently called the service implementation. Each <service> element associates a physical address with a particular
binding for a portType. For a SOAP accessible web service, the physical address would correspond to the URL where the service can be accessed. It should be clear that many different providers could implement the same service, described using the identical service interface definition (including <types>, <message>s, <portType>, and <binding>) and yet provide their own, different service implementation. Each service implementation would refer to the physical address of the particular provider’s implementation. Now, if a client had some way of searching for many different implementations of a single interface then it could pick and choose based on parameters, such as price and availability. That facility is provided by UDDI and is described in the next section. An example WSDL document appears below.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<definitions
 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
 xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:s0="http://C2B/ExampleCarItf"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/
 xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/
<types>
<s:schema elementFormDefault="qualified" targetNamespace="http://C2B/ExampleCarItf">
<s:element name="queryCars">
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="make" type="s:string" />
<s:element minOccurs="0" maxOccurs="1" name="model" type="s:string" />
<s:element minOccurs="1" maxOccurs="1" name="year" type="s:int" />
</s:sequence>
</s:complexType>
</s:element>
<s:element name="queryCarsResponse">
<s:complexType>
<s:sequence>
<s:element minOccurs="1" maxOccurs="1" name="queryCarsResult" type="s:boolean" />
<s:element minOccurs="0" maxOccurs="1" name="cs" type="s0:ArrayOfQueryCarStruct" />
</s:sequence>
</s:complexType>
</s:element>
<s:complexType name="ArrayOfQueryCarStruct">
<s:sequence>
<s:element minOccurs="0" maxOccurs="unbounded" name="queryCarStruct" type="s0:queryCarStruct" />
</s:sequence>
</s:complexType>
</s:schema>
</types>
</definitions>
```
<s:complexType>
    <s:complexType name="queryCarStruct">
        <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="vin" type="s:string" />
            <s:element minOccurs="1" maxOccurs="1" name="price" type="s:float" />
            <s:element minOccurs="1" maxOccurs="1" name="mileage" type="s:int" />
        </s:sequence>
    </s:complexType>
</s:complexType>
</s:schema>

<message name="queryCarsSoapIn">
    <part name="parameters" element="s0:queryCars" />
</message>

<message name="queryCarsSoapOut">
    <part name="parameters" element="s0:queryCarsResponse" />
</message>

<portType name="CarWSSoap">
    <operation name="queryCars">
        <documentation>Returns vin, price, and mileage for cars that match the query.</documentation>
        <input message="s0:queryCarsSoapIn" />
        <output message="s0:queryCarsSoapOut" />
    </operation>
</portType>

<portType name="CarWSHttpGet" />
<portType name="CarWSHttpPost" />

<binding name="CarWSSoap" type="s0:CarWSSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
    <operation name="queryCars">
        <input>
            <soap:body use="literal" />
        </input>
        <output>
            <soap:body use="literal" />
        </output>
    </operation>
</binding>

<binding name="CarWSHttpGet" type="s0:CarWSHttpGet">
    <http:binding verb="GET" />
</binding>

<binding name="CarWSHttpPost" type="s0:CarWSHttpPost">
    <http:binding verb="POST" />
</binding>

<service name="CarWS">
    <port name="CarWSSoap" binding="s0:CarWSSoap">
        <soap:address location="http://localhost/ws/CarWS.asmx" />
    </port>
    <port name="CarWSHttpGet" binding="s0:CarWSHttpGet">
        <http:address location="http://localhost/ws/CarWS.asmx" />
    </port>
    <port name="CarWSHttpPost" binding="s0:CarWSHttpPost">
        <http:address location="http://localhost/ws/CarWS.asmx" />
    </port>
</service>
</definitions>
7.1.3 Universal Description, Discovery, and Integration

Whereas SOAP and WSDL are standards maintained by the World Wide Web Consortium (W3C), the Universal Description, Discovery, and Integration (UDDI) specification exists as the result of cooperation between IBM, Microsoft, and Ariba. These companies realized that for web services to quickly reach the greatest potential audience, there must be some simple mechanism available for developers to share their services with other developers across the globe. UDDI helps solve this problem by providing a publicly accessible registry for companies (or any individual with a web address) to advertise their services.

![UDDI data structures](image)

**Figure 7-3. UDDI data structures**

UDDI solves two primary problems. First, it provides a specification for a set of operator nodes used to maintain business data including contact information, classification, and details regarding available services. These operator nodes replicate data across themselves to ensure that information entered at one node is available at all other nodes. At present, the two most notable UDDI operator nodes are maintained by Microsoft (at http://uddi.microsoft.com) and by IBM (at
http://www.ibm.com/services/uddi). The second issue solved by UDDI concerns programmatic access to the data maintained in the UDDI repository. Programmatic access is implemented through a set of SOAP accessible methods exposed at each UDDI operator node.

The type of information maintained at these nodes is based on the model shown in Figure 7-3. As illustrated in the figure, the top of the UDDI data hierarchy represents information about a particular business entity. To provide a “yellow pages” style search, the categoryBag field for the business entity is used to describe detailed information related to the general business functions. Currently used classification mechanisms, include the North American Industry Classification System (NAICS-1997), the Universal Standard Products and Services Codes (UNSPSC-7.03), ISO 3166 Geographic Taxonomy, and the Standard Industrial Classification (SIC-1987). Thus, if a Washington based business handles custom computer programming tasks using C++, it would store the following classifications in their categoryBag:

- NAICS 541511 Custom Computer Programming Services
- UNSPSC 81.11.16.07.00 Programming for C or C++
- ISO 3166 US-WA Washington, USA, World

Once a business entity has registered, it is then able to register information regarding any number of corresponding services that it supports. Supported service types are not limited to web services and may include items as simple as help desk phone numbers or URLs for online registration mechanisms. Whereas the businessService maintains general descriptive information pertaining to a supported service, the bindingTemplate is used to store the network endpoint address, or access point, to the service, as well as
references to detailed technical information, such as a WSDL description of the service interface. The relationship between WSDL and UDDI is made clearer in the following section, but at this juncture, it is important to realize that the UDDI registry enables the public to discover what services a company provides, as well as help to locate technical descriptions of those services. Those descriptions are likely to be WSDL interface definitions.

Table 7-1. Methods of the UDDI interfaces

<table>
<thead>
<tr>
<th>businessEntity</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>find_business</td>
<td>Inquiry</td>
</tr>
<tr>
<td>get_businessDetail</td>
<td>Inquiry</td>
</tr>
<tr>
<td>save_business</td>
<td>Publishing</td>
</tr>
<tr>
<td>delete_business</td>
<td>Publishing</td>
</tr>
<tr>
<td>businessService</td>
<td></td>
</tr>
<tr>
<td>find_service</td>
<td>Inquiry</td>
</tr>
<tr>
<td>get_serviceDetail</td>
<td>Inquiry</td>
</tr>
<tr>
<td>save_service</td>
<td>Publishing</td>
</tr>
<tr>
<td>delete_service</td>
<td>Publishing</td>
</tr>
<tr>
<td>bindingTemplate</td>
<td></td>
</tr>
<tr>
<td>find_binding</td>
<td>Inquiry</td>
</tr>
<tr>
<td>get_bindingDetail</td>
<td>Inquiry</td>
</tr>
<tr>
<td>save_binding</td>
<td>Publishing</td>
</tr>
<tr>
<td>delete_binding</td>
<td>Publishing</td>
</tr>
<tr>
<td>tModel</td>
<td></td>
</tr>
<tr>
<td>find_tModel</td>
<td>Inquiry</td>
</tr>
<tr>
<td>get_tModelDetail</td>
<td>Inquiry</td>
</tr>
<tr>
<td>save_tModel</td>
<td>Publishing</td>
</tr>
<tr>
<td>delete_tModel</td>
<td>Publishing</td>
</tr>
</tbody>
</table>

After a company has stored its business information in UDDI, the company must provide other companies with a convenient mechanism to search for and extract information regarding those services. UDDI provides an API, accessible as a SOAP-based web service, exposed at each operator node. The methods available on the web service correspond very closely to the data model just discussed. In general, each of the
core UDDI data structures (businessEntity, businessService, etc) is accessible through a method. Different methods implement operations such as find, get details, save, and delete. The table below outlines most of the important UDDI methods and distinguishes the Inquiry API from the Publishing API. An HTML Form-based interaction method also exists as an alternative to the programmatic interface described in the table.

7.1.4 Using WSDL in a UDDI Registry

We have described the three primary technologies involved in the development of web services: SOAP, WSDL, and UDDI. However, the relationship between these technologies, and specifically between WSDL and UDDI, still requires additional clarification. UDDI was developed as a very general specification to be used with any service description mechanism. For this reason, the techniques employed to properly register WSDL descriptions in a UDDI registry are not immediately apparent from looking at the two standards in isolation.

The diagram in Figure 7-4 helps to sum up the process. The WSDL interface document is referenced by the <overviewURL> element of a tModel. This tModel is then referenced, by its tModelKey, from a <bindingTemplate> element nested inside of a <businessService>. The <bindingTemplate> element also contains an <accessPoint> element that points to the network access endpoint of the service described in the tModel.
7.2 Two Different Architectures

Applications built using SOAP, WSDL, and UDDI often conform to what is called the Service-Oriented Architecture (SOA). As illustrated in Figure 7-5, a system based on SOA has three major components: the service registry, the service consumer, and the service provider. The service registry is usually realized as a UDDI operator node. Service providers register WSDL descriptions of their services with the service registry. Later on, service consumers discover these services by performing searches on the UDDI operator cloud. Once a service is located, the service consumer extracts the WSDL service description, including both the service interface and service implementation and use it to generate code for accessing the service. In almost all cases, the code generation is performed at design time. This is necessary since the service description requires a human intellect to understand the required message parameters and ensure that the service consumer’s code provides the appropriate arguments to the service provider’s interface. Any changes to the service provider’s interface requires, in addition to re-

Figure 7-4. Mapping WSDL into UDDI
registration with the service registry, a recompile (or redesign) at each service consumer’s site.

Service Oriented Architecture

![Service Oriented Architecture Diagram](image)

To combat both the design time dependence and interface fragility problems inherent in SOA, we propose the content-oriented architecture (COA) illustrated in Figure 7-6. The similarities between SOA and COA are not accidental. SOA, and the technologies used to support it, form the foundation for COA. In COA, the service registry has been replaced with a matchmaker (described in Chapter 6). In addition, a new component, the type repository, has been added. The type repository will be implemented as the Global Type Repository described throughout this work and will contain the specification of a content-biased language (CBL). The service consumer and service provider have been replaced with peer-to-peer constituent system agents. This supports an important goal of C2B, which is placing the consumer on equal footing with the business.
When the constituent system agents in COA wish to communicate, they do so using the CBL contained in the GTR. If the two agents speak different dialects then messages can be routed through the matchmaker. Using cross-dialect mappings (PDG mappings) from the GTR, the matchmaker can provide a translation service to the agents to increase the probability of a successful communication.

Notice that some lines in Figures 7-5 and Figure 7-6 are dotted, while others are solid. The dotted lines represent activities occurring at design time, while the solid lines represent run-time actions. Through addition of the type repository, it is now possible to make all activities supported by SOA function at run-time. In this way, COA provides a top layer that is currently missing in the SOA hierarchy. The comparison is illustrated in
Figure 7-7. The acronym CDL designates the content description language that is introduced towards the end of this chapter.

<table>
<thead>
<tr>
<th>SOA Interop Stack</th>
<th>Undefined Content Biased Communication (CDL)</th>
<th>COA Interop Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDDI</td>
<td>GTR and Matchmakers</td>
<td></td>
</tr>
<tr>
<td>Endpoint Access Protocol (SOAP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eXtensible Markup Language (XML)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet Protocols (HTTP, TCP/IP)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7-7. Interop stacks: adapted from the UDDI technical white paper [128]

One final important point concerns the style of interfaces in COA. Whereas an SOA interface is generally composed of arbitrary, multiple argument method calls, a COA interface could consist of a fixed set of well-known single argument method calls. A good analogy is human communication. A human has only one set of ears (one interface) that handles all audible communication. Once the communication has entered the brain (agent), it is dispatched to the appropriate handler based on the message content. Thus the COA, and its associated CBL, clearly represent a more flexible, anthropomorphic model of software design. Now, if an agent design is changed to require different information (different arguments), the calling agent does not need to be recompiled. Instead, the new callee version simply asks the caller for the additional information. Given the existence of a CBL, this request is unambiguous and may succeed though it had no previous precedent. Furthermore, if the caller cannot immediately supply the additional information, an ancillary GUI agent could be spawned to solicit help from a human operator.
7.3 SOA-Based Web Service

In the remainder of this chapter, we compare the functionality of an SOA based system with a similarly functional COA based system. Our example is based on a modified version of the Car-Ownership PBU described throughout the latter part of Chapter 5. In particular, a car dealer service is created for use by potential car buyers. The SOA model of the Car-Dealer system components appears in Figure 7-8.

The car dealer plays the role of the service provider; the service consumer is realized as a private buyer; and the service broker is implemented as a UDDI operator cloud. The following WSDL document describes the details of the CarDealer interface and its associated methods.
<s:schema elementFormDefault="qualified" targetNamespace="http://C2B/ExampleCarItf">
  <s:element name="queryCars">
    <s:complexType>
      <s:sequence>
        <s:element minOccurs="0" maxOccurs="1" name="make" type="s:string" />
        <s:element minOccurs="0" maxOccurs="1" name="model" type="s:string" />
        <s:element minOccurs="1" maxOccurs="1" name="year" type="s:int" />
      </s:sequence>
    </s:complexType>
  </s:element>

  <s:element name="queryCarsResponse">
    <s:complexType>
      <s:sequence>
        <s:element minOccurs="1" maxOccurs="1" name="queryCarsResult" type="s:boolean" />
        <s:element minOccurs="0" maxOccurs="1" name="cs" type="s0:ArrayOfQueryCarStruct" />
      </s:sequence>
    </s:complexType>
  </s:element>

  <s:complexType name="ArrayOfQueryCarStruct">
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="unbounded" name="queryCarStruct" type="s0:queryCarStruct" />
    </s:sequence>
  </s:complexType>

  <s:complexType name="queryCarStruct">
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1" name="vin" type="s:string" />
      <s:element minOccurs="1" maxOccurs="1" name="price" type="s:float" />
      <s:element minOccurs="1" maxOccurs="1" name="mileage" type="s:int" />
    </s:sequence>
  </s:complexType>

  <s:element name="queryCarDetail">
    <s:complexType>
      <s:sequence>
        <s:element minOccurs="0" maxOccurs="1" name="vin" type="s:string" />
      </s:sequence>
    </s:complexType>
  </s:element>

  <s:element name="queryCarDetailResponse">
    <s:complexType>
      <s:sequence>
        <s:element minOccurs="1" maxOccurs="1" name="queryCarDetailResult" type="s:boolean" />
        <s:element minOccurs="0" maxOccurs="1" name="make" type="s:string" />
        <s:element minOccurs="0" maxOccurs="1" name="model" type="s:string" />
        <s:element minOccurs="1" maxOccurs="1" name="year" type="s:int" />
        <s:element minOccurs="1" maxOccurs="1" name="mileage" type="s:int" />
        <s:element minOccurs="1" maxOccurs="1" name="price" type="s:float" />
        <s:element minOccurs="0" maxOccurs="1" name="color" type="s:string" />
        <s:element minOccurs="1" maxOccurs="1" name="tranny" type="s0:transmissionType" />
        <s:element minOccurs="1" maxOccurs="1" name="numDoors" type="s:int" />
      </s:sequence>
    </s:complexType>
  </s:element>

  <s:simpleType name="transmissionType">
    <s:restriction base="s:string">
      <s:enumeration value="Automatic" />
    </s:restriction>
  </s:simpleType>
</s:schema>
<s:element name="purchaseCar">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1" name="name" type="s:string" />
      <s:element minOccurs="0" maxOccurs="1" name="addr" type="s:string" />
      <s:element minOccurs="0" maxOccurs="1" name="phone" type="s:string" />
      <s:element minOccurs="0" maxOccurs="1" name="creditType" type="s:string" />
      <s:element minOccurs="0" maxOccurs="1" name="creditExp" type="s:string" />
      <s:element minOccurs="0" maxOccurs="1" name="creditNum" type="s:string" />
    </s:sequence>
  </s:complexType>
</s:element>

<s:element name="purchaseCarResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="purchaseCarResult" type="s:boolean" />
      <s:element minOccurs="1" maxOccurs="1" name="transactionStatus" type="s:int" />
      <s:element minOccurs="0" maxOccurs="1" name="confirmationNumber" type="s:string" />
    </s:sequence>
  </s:complexType>
</s:element>

<s:schema>
  <types>
    <message name="queryCarsSoapIn">
      <part name="parameters" element="s0:queryCars" />
    </message>
    <message name="queryCarsSoapOut">
      <part name="parameters" element="s0:queryCarsResponse" />
    </message>
    <message name="queryCarDetailSoapIn">
      <part name="parameters" element="s0:queryCarDetail" />
    </message>
    <message name="queryCarDetailSoapOut">
      <part name="parameters" element="s0:queryCarDetailResponse" />
    </message>
    <message name="purchaseCarSoapIn">
      <part name="parameters" element="s0:purchaseCar" />
    </message>
    <message name="purchaseCarSoapOut">
      <part name="parameters" element="s0:purchaseCarResponse" />
    </message>
  </types>
  <portType name="CarWSSoap">
    <operation name="queryCars">
      <documentation>Returns vin, price, and mileage for cars that match the query.</documentation>
      <input message="s0:queryCarsSoapIn" />
      <output message="s0:queryCarsSoapOut" />
    </operation>
    <operation name="queryCarDetail">
      <documentation>Returns car detail for a particular VIN.</documentation>
      <input message="s0:queryCarDetailSoapIn" />
      <output message="s0:queryCarDetailSoapOut" />
    </operation>
    <operation name="purchaseCar">
    </operation>
  </portType>
</s:schema>
<documentation>Completes a sale.</documentation>
<input message="s0:purchaseCarSoapIn" />
<output message="s0:purchaseCarSoapOut" />
</operation>
</portType>
<binding name="CarWSSoap" type="s0:CarWSSoap">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
<operation name="queryCars">
<input>
<soap:body use="literal" />
</input>
<output>
<soap:body use="literal" />
</output>
</operation>
<operation name="queryCarDetail">
<soap:operation soapAction="http://C2B/ExampleCarItf/queryCarDetail" style="document" />
<input>
<soap:body use="literal" />
</input>
<output>
<soap:body use="literal" />
</output>
</operation>
<operation name="purchaseCar">
<input>
<soap:body use="literal" />
</input>
<output>
<soap:body use="literal" />
</output>
</operation>
</binding>
<service name="CarWS">
<port name="CarWSSoap" binding="s0:CarWSSoap">
<soap:address location="http://localhost/ws/CarWS.asmx" />
</port>
</service>
</definitions>

The developer of a service consumer application would find the WSDL file presented above by searching through the UDDI operator cloud. Using a tool, such as wsdl.exe, provided as part of the Microsoft .NET SDK, the developer of the service consumer application could automatically create a C# wrapper class containing stubbed implementations of each method in the WSDL file. The stubbed methods would implement the required marshalling code to make the actual calls against the web service.
From the perspective of the service consumer application, these stubbed methods would behave just the same as any local methods – ignoring performance differences.

An alternative approach to using wsdl.exe, is to manually code the SOAP calls against the web service. This technique allows the developer to become more intimate with the details of SOAP, and, one may argue, helps to move away from the RPC mentality. The code required to perform the manual steps is reasonably simple. An excerpt from a VB6 program used to call the queryCars method follows:

```vbnet
Private Sub queryCars_Click()
    Dim sb As String
    Dim ox As XMLHTTP
    Dim od As DOMDocument

    Set ox = New XMLHTTP
    ox.open "POST", "http://localhost:8080/ws/CarWS.asmx", False

    sb = "" & _
        "<soap:Envelope" & _
        " xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"" & _
        " xmlns:xsd = "http://www.w3.org/2001/XMLSchema" & _
        " xmlns:soap = "http://schemas.xmlsoap.org/soap/envelope/"" & _
        "<soap:Body" & _
        "<queryCars xmlns=""http://C2B/ExampleCarItf"">" & _
        "<make>" & carMake & "</make>" & _
        "<model>" & carModel & "</model>" & _
        "<year>" & carYear & "</year>" & _
        "</queryCars>" & _
    "</soap:Body>" & _
    "</soap:Envelope>"

    ox.setRequestHeader "Content-Type", "text/xml"
    ox.setRequestHeader "SOAPAction", "http://C2B/ExampleCarItf/queryCars"

    ox.send sb

    Set od = ox.responseXML
    ' Display the response in a text box
    soapResult.Text = od.xml

End Sub
```

The primary problem with either the manual approach, or use of wsdl.exe, is that the arguments to the method call, or contents of the soap:Body element are specific to the particular service provider. The developer of the service consumer application must use
comments provided in the WSDL to discern the meaning of each parameter. In reality, the WSDL description could be broken into two portions, one for interface (containing the types, messages, portTypes, and bindings) and one for implementation (containing the service element). Doing so allows the service interface to be registered as a UDDI tModel, and promotes interface reuse, though it only minimally alleviates the developer’s burden of having to interpret method parameters. This technique also enables support for runtime implementation selection but still falls short of providing a means to dynamically switch interfaces. To realize the latter goal, the <types> section of the WSDL document would have to be standardized, and that is exactly the purpose of the Global Type Repository and the province of PDGs.

### 7.4 COA-Based System

The Content-Oriented Architecture represents the next phase of software system design. In particular, COA supports dynamic changes to interfaces and makes no demands for interface sharing as a means to enable reuse. In addition, it provides a well-defined mechanism for sharing parameter types across disparate interfaces. As a result, constituent system developers are free to design their interfaces and methods in any way they choose.

This flexibility will greatly enhance the utility of web services and will provide eCommerce opportunities for individuals and corporations alike. Furthermore, application developers will no longer avoid adding new functionality for fear of breaking existing clients. Web service implementations will morph into dynamic agent-based communities whose participants can engage in open dialog with both man and machine. Clerical errors and miscommunication will become a thing of the past as individuals carry around accurate personal information in an easily accessible and ultimately
extensible format. And, if we’re lucky, no one will ever ask us to fill out another form again.

Figure 7-9. Car-dealer PDG

7.4.1 Converting a PDG to a CDL-Schema

The PDG representation described in section 5.9 served well for explanatory purposes, but it is an unlikely candidate for direct input into a computer. What is required is a new representation containing the same information provided by a PDG, but declared in a more computer friendly format. The eXtensible Markup Language (XML) is an excellent tool, perfectly suited to this task. The following XML document represents the PDG illustrated in Figure 7-9. The document is an example of our proposed Content
Description Language Schema (CDL-Schema) and serves as the basis of the discussion on COA-based systems.

```xml
<CDLSchema
  xmlns:xsi="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://schemas.c2b.org/Car-Dealer.cdlsls/"
  targetNamespace="http://schemas.c2b.org/Car-Dealer.cdlsls/"
  elementFormDefault="qualified">

  <types>
    <xs:schema elementFormDefault="qualified"
      targetNamespace="http://schemas.c2b.org/Car-Dealer.cdlsls/">
      <xs:simpleType name="enumTransmissionType">
        <xs:restriction base="xs:string">
          <xs:enumeration value="automatic"/>
          <xs:enumeration value="manual"/>
        </xs:restriction>
      </xs:simpleType>

      <xs:simpleType name="enumColorType">
        <xs:restriction base="xs:string">
          <xs:enumeration value="blue"/>
          <xs:enumeration value="green"/>
          <xs:enumeration value="red"/>
          <xs:enumeration value="white"/>
          <xs:enumeration value="black"/>
        </xs:restriction>
      </xs:simpleType>

      <xs:simpleType name="enumDoorType">
        <xs:restriction base="xs:string">
          <xs:enumeration value="frontDriver"/>
          <xs:enumeration value="frontPassenger"/>
          <xs:enumeration value="rearDriver"/>
          <xs:enumeration value="rearPassenger"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:schema>
  </types>

  <language>
    <defSimpleConcepts>
      <simpleConcept name="transmissionType" id="1"
        creator="http://www.c2b.org/creators/autotech.cmp">
        <NUID/>
        <type ref="tns:enumTransmissionType"/>
        <contexts>
          <ctx name="tns:Car"/>
    ```
<simpleConcept name="make" id="2" creator="http://www.c2b.org/creators/autotech.cmp">
  <NUID/>
  <type ref="xs:string"/>
  <contexts>
    <ctx name="tns:Car"/>
  </contexts>
</simpleConcept>

<simpleConcept name="model" id="3" creator="http://www.c2b.org/creators/autotech.cmp">
  <NUID/>
  <type ref="xs:string"/>
  <contexts>
    <ctx name="tns:Car"/>
  </contexts>
</simpleConcept>

<simpleConcept name="year" id="4" creator="http://www.c2b.org/creators/autotech.cmp">
  <NUID/>
  <type ref="xs:year"/>
  <contexts>
    <ctx name="tns:Car"/>
  </contexts>
</simpleConcept>

<simpleConcept name="price" id="5" creator="http://www.c2b.org/creators/autotech.cmp">
  <NUID/>
  <type ref="xs:decimal"/>
  <contexts>
    <ctx name="tns:Car"/>
  </contexts>
</simpleConcept>

<simpleConcept name="mileage" id="6" creator="http://www.c2b.org/creators/autotech.cmp">
  <NUID/>
  <type ref="xs:int"/>
  <contexts>
    <ctx name="tns:Car"/>
  </contexts>
</simpleConcept>

<simpleConcept name="color" id="7" creator="http://www.c2b.org/creators/autotech.cmp">
  <NUID/>
  <type ref="tns:enumColorType"/>
  <contexts>
    <ctx name="tns:Car"/>
  </contexts>
</simpleConcept>

<simpleConcept name="powerWindow" id="8" creator="http://www.c2b.org/creators/autotech.cmp">
  <NUID/>
  <type ref="xs:boolean"/>
  <contexts>
    <ctx name="tns:Door"/>
  </contexts>
</simpleConcept>
<simpleConcept name="name" id="9" creator="http://www.c2b.org/creators/autotech.cmp">
    <NUID/>
    <type ref="xs:string"/>
    <contexts>
        <ctx name="tns:Person"/>
    </contexts>
</simpleConcept>

<simpleConcept name="address" id="10" creator="http://www.c2b.org/creators/autotech.cmp">
    <NUID/>
    <type ref="xs:string"/>
    <contexts>
        <ctx name="tns:Person"/>
    </contexts>
</simpleConcept>

<simpleConcept name="phone" id="11" creator="http://www.c2b.org/creators/autotech.cmp">
    <NUID/>
    <type ref="xs:string"/>
    <contexts>
        <ctx name="tns:Person"/>
    </contexts>
</simpleConcept>

<simpleConcept name="type" id="12" creator="http://www.c2b.org/creators/autotech.cmp">
    <NUID/>
    <type ref="xs:string"/>
    <contexts>
        <ctx name="tns:Credit-Card"/>
    </contexts>
</simpleConcept>

<simpleConcept name="expiration" id="13" creator="http://www.c2b.org/creators/autotech.cmp">
    <NUID/>
    <type ref="xs:date"/>
    <contexts>
        <ctx name="tns:Credit-Card"/>
    </contexts>
</simpleConcept>

<simpleConcept name="number" id="14" creator="http://www.c2b.org/creators/autotech.cmp">
    <NUID/>
    <type ref="xs:string"/>
    <contexts>
        <ctx name="tns:Credit-Card"/>
    </contexts>
</simpleConcept>

<simpleConcept name="dealer" id="15" creator="http://www.c2b.org/creators/autotech.cmp">
    <type ref=""/>
    <contexts>
        <ctx name="tns:Car-Dealer"/>
    </contexts>
</simpleConcept>
The CDL-Schema presented above represents information pertinent to a car dealer. For demonstrative purposes, we have kept the example schema reasonably small. The complete markup for CDL-Schema has a number of additional features as highlighted by the following pseudo-XML Schema for CDL-Schema.
7.4.2 Basic Architecture

A COA-based system depends on the components identified in Figure 7-6. A more detailed explanation of the purpose and functionality of each of these components was provided in the previous chapter. In this section, we shift focus to examine lower level design features of each component. It is important to keep in mind that no particular implementation is being implied. Rather, the goal of this section is to help illuminate the overall functionality of the system by providing additional insight into the interactions of the various components.

Figure 7-10 shows a layered design for the Global Type Repository (GTR), a matchmaker, and two constituent systems. The constituent system models serve also to demonstrate the expected behavior of the constituent system agents. In reality, the agents may be itinerant and, therefore, separate from their constituent system. However, no particular division of functionality is imposed between the agents and their containing system so we are free to combine the two for the purposes of exposition.

The GTR contains the registered EOOTS from all constituent systems in the global system. In addition, the GTR contains mappings between overlapping EOOTS representations. The functionality of the GTR is that of a passive information repository. Using a standard interface, such as the web service illustrated in diagram, constituent systems can register, update, and search for EOOTS. The GTR also exposes additional
interfaces to enable addition, retrieval, and modification of EOOTS mappings. It should be noted that the GTR could be implemented using COA instead of SOA. However, it would be far more difficult to provide the initial description of a new technology in terms of itself so we have opted to show how existing and familiar technologies could be leveraged to produce the required behavior.

![Layered architectures for global system components](image)

The matchmaker represents the central point of all global system communication. As described previously, the matchmaker exposes a semi-permeable membrane that selectively accepts pellets and agents to interact under its control. This membrane is shown in the diagram as a web service exposing methods including registerPellet(), removePellet(), searchPellet(), registerAgent(), and removeAgent(). In addition, the
matchmaker offers the participants a number of valuable services. Outside of a secure and monitored execution environment, the most important service supplied by the matchmaker is the role it will play as a distributed transaction coordinator. This functionality is illustrated in the diagram using the acronym (DTC) and may be exposed as a web service supporting methods such as join(), start(), commit(), and egress().

The final COA component we detail in this section is the constituent system along with its agents. In Figure 7-10, two constituent systems were illustrated for consistency with the COA illustration of Figure 7-6. As expected in a C2B system, both constituent system participants have the same underlying structure, further emphasizing the equality of all participants, corporate or individual. All agents, regardless of their purpose, can be implemented as a web service with only two methods. The first method, ask(), is called by other transaction participants when they require specific content from the receiver agent. The ask() method accepts a question in the form of a content-biased language (CBL) as described in the GTR. Using the Open Dispatch Table (ODT), the dispatcher module parses the CBL and passes the message to the appropriate handler function. The handler function does any required processing and uses the Constituent System Data to provide an answer to the caller.

The scenario just described represents the most basic form of inter-agent communication. Consider now an ask() call from agent $A_1$ to agent $A_2$. We now allow the possibility that $A_1$ speaks a different dialect than $A_2$, and furthermore we permit dynamic changes to the information required by $A_2$. The following steps will occur to ensure a successful transaction:
1. Agent A1 acts as an initiating participant and calls the matchmaker’s start() method.

2. The matchmaker calls A1::start() and A2::start() thereby beginning the transaction.

3. A1 calls A2::ask() by passing a CBL statement in dialect L1.

4. The matchmaker intercepts the ask() message and converts from dialect L1 to dialect L2 based on the mappings made available in the GTR.

5. A2 receives the ask() message in L2.

6. The dispatcher in A2 reads its ODT and sends the request message to the appropriate handler.

7. The handler realizes that necessary information is missing from the request.

8. The handler initiates a callback to A1.


10. The matchmaker intercepts the ask() message and converts from dialect L2 to dialect L1 based on the mappings made available in the GTR.


12. The dispatcher in A1 reads its ODT and sends the request message to the appropriate handler.

13. The handler attempts to find or generate the requested information. If the information is not available, an ancillary GUI agent may be sent to a human operator. Assume the information is obtained using one of these methods.

14. The handler forms a CBL response in dialect L1 and sends it out.

15. The matchmaker intercepts the response, converts it, and passes it along to A2.

16. The dispatcher in A2 sends the new information back to the original handler.

17. The handler completes the request and sends a response to A1.

### 7.4.2.1 Open dispatch tables

The scenario described above elucidates many interesting design details. The open dispatch table (ODT) could be implemented as an XML document that provides, in
addition to the dispatch information required by the dispatcher, an externally accessible
description of expected method arguments. This technique recalls the RPC-like nature of
current web services but supplements them with additional semantic information. One
reason for taking this approach would be to eliminate the back-and-forth communication
implied by the scenario above. Of course, another option would be to cache the details of
the method call after they are first discovered, and then base the format of all future calls
on those details. If the information required by the method ever changes, a callback can
be expected from the receiver, and the cache details can be updated at that time. What is
ultimately important is that the receiver can change the requirements of its interfaces
without negatively impacting the callers. This is certainly possible down to the level of
the dispatcher, but further down within the code of the function handlers this requirement
may prove more challenging. One possible approach would be to use a technique similar
to the structured exception handling (SEH) provided by C++. Specifically, when the
handler attempts to access a “method parameter” that was not sent by the caller, an
exception is thrown, and when the information is later obtained, an
EXCEPTION_CONTINUE_EXECUTION occurs. At the implementation level, this
may done by storing all program variables in a form of shared memory, using
getters/setters for all parameter access, and maintaining two threads, one for
implementing the handler and another for assuring the handler has the data it requires.

7.4.2.2 Event-driven open multiple predicate dispatch

Another interesting detail of the proposed design relates to the functionality of the
dispatcher. Whereas a dispatcher in an object-oriented language generally only considers
the runtime type of the receiver object (and the static types of the method parameters), the
dispatcher in an agent could provide functionality more closely related to the predicate
dispatching described by Ernst, Kaplan, and Chambers [41]. We have coined the term *Event-Driven Open Multiple Predicate Dispatch* (ED-OMPD) to describe the expected behavior of COA dispatching. Within the context of a multi-participant transaction, an ask() call may be propagated to all members of the transaction. That is, if agent A₁ wants information I₁, it can simply send the request to the matchmaker, without specifying any particular receiver agent. The matchmaker will dispatch the message to all participants (multiple dispatch), or to a subset of those participants based on some predicate over the content of the information request (predicate dispatch). After the matchmaker dispatch, a further, more detailed level of dispatch could be performed in the agent. It is interesting to note that the dispatch module may now reside in the matchmaker, the agents, or both. In essence, the ask() method acts as an event to the transaction and is dispatched based on the ODTs of the various participant agents and the constraints imposed by the matchmaker.

**7.4.3 Use Case**

In this section, we present a use case based on the Car-Dealer PDG illustrated in Figure 7-9 of Section 7.4.1. In that section, we also provided the CDL-Schema corresponding to the Car-Dealer EOOTS. This hypothetical schema provides the language used for a car purchase transaction within the global system. The steps detailed below follow the primary tasks involved in establishing a constituent system, sewing wild EOOTS, registering a pellet, initiating a transaction, engaging in a transaction, and finally completing a transaction. Throughout the process, we examine the flow of some important messages, as well as the content of those messages. In the end, the general utilization and the advantages of a COA system should be clear.
7.4.3.1 Creating a constituent system

The first step in creating a constituent system is establishing a web presence. This does not necessarily mean obtaining an IP address, as a virtual directory is adequate. What is required is a physical network endpoint with at least intermittent network connectivity. Once the web address has been established, the constituent system owner must obtain a unique identification number for his system. This number will be used for many purposes, one of which is to populate the creator attribute for any wild EOOTS created by the owner of this constituent system. Once the EOOTS have been sewn, other global system users may choose to ignore modifications to the EOOTS made by anyone other than the original creator. For now, it is useful to imagine the identifier as having the form of a URI such as http://www.c2b.org/creators/autotech.cmp. At this point, the constituent system is created and is known to the global system, though it has not yet participated in any global system activities.

7.4.3.2 Sewing wild EOOTS

Once a constituent system has been created, its owners may require additions to the content-biased language (CBL) currently available in the Global Type Repository (GTR). Best practices suggest that the developers first look through the GTR to see if satisfactory content descriptions have already been created; if so, the developers should use such resources to the greatest extent possible. For the purposes of this example, it is useful to assume that no satisfactory vocabulary exists and that the constituent system designers must create their own. The first step in this process is designing a PDG. There is no doubt that a sophisticated GUI tool could greatly simplify this process and also help to automate the conversion of the PDG into the corresponding CDL-Schema. As mentioned
above, both the PDG and the CDL-Schema for the current example were presented in section 7.4.1.

Once the CDL-Schema document has been created, it must be added to the GTR. This process is called sewing your wild EOOTS. In general, there are two steps that must be completed to successfully sew your EOOTS. First, the new CDL-Schema must be added to the GTR, and second, the new EOOTS must be mapped to other existing EOOTS in the system. These mappings help ensure successful communication even when participants speak different dialects of the CBL. The programmatic interface to the GTR exposes the methods necessary to perform these two steps, and one may imagine the following calls:

    RegisterEOOTS("http://schemas.c2b.org/Car-Dealer.cdls")
    AddMapping("http://schemas.c2b.org/Car-Dealer-to-Car-Ownership.cdlm")

7.4.3.3 Registering a pellet

It is important to note that the Car-Dealer schema could be used by either a company or, with minor modification, by a person, to advertise information about a car for sale. The change required to enable use by an individual is to update the Car-Dealer/sells/from reference node to reference either a person or a dealer. This may be easily accomplished through addition of a subject node called “seller,” that serves as a base type for both person and dealer. To help keep this example closer to the SOA example of section 7.3, this change has not been made.
Given that the EOOTS has now been registered, the next step in the process is to send a pellet into the global system. In the case of a car dealer, there are two different pellet types that would provide useful functionality. First, a service provision pellet could be created to help advertise the fact that the car dealer sells cars to people. The form of this pellet is illustrated in Figure 7-11, and the CDL representation follows.

```xml
<CDL
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:CDLSchema="http://schemas.c2b.org/Car-Dealer.cdls/"
xmlns:tns="http://schemas.c2b.org/Car-Dealer.cdls/"
elementFormDefault="qualified">

<language>

<refSimpleConcepts>

<simpleConcept name="dealer" id="15">
    <GUID href="http://www.c2b.org/creators/autotech.cmp"/>
</simpleConcept>

</refSimpleConcepts>

<refRelationships>

<relationship name="sells" id="18" creator="http://www.c2b.org/creators/autotech.cmp">
    <intended as="provider"/>
</relationship>

</refRelationships>

</CDL>
```
The content description language (CDL) listing shown above is a subgraph of the original CDL-Schema. However, whereas the CDL-Schema defined the types of the simpleConcepts and the pools for the GUIDs, the CDL document provides values for the simple types and specific entries from the GUID pools. In the example, the particular company providing the car dealer service is clearly identified by the GUID http://www.c2b.org/creators/autotech.cmp.

The second type of pellet a car dealer would export provides factual information about a particular car being sold. Imagine that the dealer has a 2002 Chevrolet Corvette for sale. The associated pellet would appear as shown in Figure 7-12 and would be described by the CDL listing below.
Once these pellets have been created, the constituent system for the car dealer needs to inject them into the global system and ensure a matchmaker accepts them. To perform this registration, the constituent system could perform the calls against a specific matchmaker. Another registration alternative would be to have an agent carry these pellets throughout the global system until a satisfactory matchmaker is located and a successful negotiation for entry is completed. In either case the calls would appear as follows:
RegisterPellet(“http://schemas.c2b.org/Car-Dealer-Provider.cdl”)

RegisterPellet(“http://schemas.c2b.org/Car-Dealer-Fact-Sell-Corvette.cdl”)

Figure 7-12. Factual information pellet providing details about a car for sale

7.4.3.4 Initiating a transaction

Once the pellets have been registered, the constituent system owner needs to wait until an interested buyer discovers the information. Within the matchmaker, it is unlikely that the pellet content will be stored using CDL. This would be equivalent to storing a telephone book in a natural language; something that is simply not done. Instead, the matchmaker will parse the information and store it in a database. This process can be automated by mapping the relational schema to the CDL-Schema corresponding to the
document. This mapping can either be to the same CDL-Schema as the input CDL document or to any other CDL-Schema that has already been mapped to the former. Either way, once the information has been stored in a database, it can be queried using relational database techniques or a CDL-based query language. An example of a CDL-based query may appear as follows:

Does Car-Dealer [id= “http://www.c2b.org/creators/autotech.cmp”] have a Chevrolet Corvette for sale?

```
USING SCHEMA http://schemas.c2b.org/Car-Dealer.cdls AS CarSchema
YNQ
alias car as CarSchema://Car-Dealer/car
alias dealer as CarSchema://Car-Dealer/dealer[id= "http://www.c2b.org/creators/autotech.cmp"]
alias sells as CarSchema://Car-Dealer/sells
WHERE
    car.make = Chevrolet
    car.model = Corvette
    sells.from ~ dealer
    sells.what ~ car
```

This Yes/No Query (YNQ) will return “yes” if the specified car dealer has registered a pellet stating that he has the requested car for sale. In our example, such a pellet was registered, and the agent that submitted the query will decide to pursue a transaction for the purpose of purchasing the automobile. This agent becomes the initiating participant in the transaction by calling the matchmaker’s start() method. Upon receipt of this method call, the matchmaker establishes a transaction ID and calls back to the start() methods of both the initiating agent and the car dealer agent. At this point, the transaction has been initiated, and the parties will attempt to close a deal.

### 7.4.3.5 Engaging in a transaction

Once the transaction begins, the participant agents must decide what content they require and what functionality they need to perform. Since the initiating agent in our example started a transaction based on the availability of a car for sale, it makes sense that the two participants will enter into a dialog concerning the details of the car. To
begin with, the initiating agent may desire additional information about the mileage, price, color, and other attributes of the Corvette. Instead of using a pre-defined interface, the initiating agent will simply ask(), the car dealer agent, for the desired information. The form of the question may be a CDL document with question marks placed in the desired value elements, or it may be expressed using a CDL query language.

An example of the former representation is illustrated in Figure 7-13. A more interesting version based on the CDL query representation is shown below.

```
USING SCHEMA http://schemas.c2b.org/Car-Ownership.cdls AS CarSchema
MVQ
alias car as CarSchema://Car-Ownership/car
WHERE
  car.VIN = "http://www.c2b.org/GUIDS/Car.VIN/ABC1234DE4354FGD5642"
RETURN
  car.price
  car.mileage
  car.color
```

The multi-valued query (MVQ) shown above is based on the Car-Ownership PDG from Chapter 5. For the purposes of this example, we have extended that schema to include price, mileage, and color in the context of car. Furthermore, it is assumed that a mapping exists between the Car-Ownership schema and the Car-Dealer schema. Provided these two conditions hold true, the query above, though not in the dialect spoken by the car dealer, will still serve well to help the initiating agent acquire the desired information. We have now achieved one degree of stretchability.

After the more detailed information has been returned to the initiating agent, it decides it would like to purchase the car and issues a commit() to the matchmaker. The matchmaker passes this commit request to the car dealer agent, but the car dealer now requires some information from the buyer. In particular, the dealer needs credit card data. Using a query similar to that shown above, the dealer asks the buyer for the needed
information. Assuming the initiating agent has that information available, it returns the
data, and the transaction is committed. On the other hand, if the initiating agent does not
have immediate access to the data, it can spawn an ancillary GUI agent to request the
information from a human user. While the system waits for a response from the human
user, the matchmaker may serialize the transaction. When the response finally arrives,
the matchmaker can resurrect the transaction and the commit can then be attempted.

![Diagram]

Figure 7-13. Using a PDG to request content

One final point is of particular interest. In the service-oriented architecture, the dealer
agent would have registered an interface with a method signature containing a parameter
for credit card information. If, at some point in the future, the dealer realized he also
needed the customer’s address and phone number, then the interface would have to be
changed. Since all clients would have been hard-coded against this interface at design
time, this change would have a profound negative impact. On the other hand, in the
content-oriented architecture, a late decision to acquire additional information would
simply entail a change in the dealer’s request. Even though the buyer agent was not
previously asked for the information, it can still easily provide the answer. As long as the
dealer’s request is expressed using a content-biased language, the buyer will understand
the request and either return the data immediately or ask a human user to provide the
required information. And, as shown above, the request need not even be in the same
dialect as spoken by the buyer. This ability, or stretchability, is what separates COA
from SOA and is what ultimately enables a consumer-to-business system.
CHAPTER 8
CONCLUSION AND FUTURE WORK

The natural languages employed in human communications represent the pinnacle of flexibility and extensibility for the transfer of information. Unfortunately, they are also often wrought with problems of ambiguity and misinterpretation. Using natural language as a basis, and remaining keenly aware of its potential pitfalls, we have described a new communication model based on what we have termed content-biased language (CBL). A CBL consists of a well-defined set of concepts and is capable of expressing relationships and properties of those concepts. The representation of a CBL as a set of EOOTS in a GTR, enables extensibility, and also ensures flexibility as different dialects can be used interchangeably through the creation of appropriate mappings.

The requirements of a C2B system could be satisfied with nothing less than the flexibility of content-biased communication. The sheer number of potential subject areas, and the multitude of possible, and meaningful communication patterns made it absolutely mandatory to establish a more flexible, more stretchable form of content transfer. This notion of stretchability, and in particular its implications for content transfer, provided the primary motivation for the ideas presented throughout this dissertation. Once this underlying problem was identified, the enabling research on perspective domain graphs, external open ontological type systems, and ultimately the content-oriented architecture could be developed.

The introduction of the content-oriented architecture as the culmination of the research helped to clearly relate the many facets of the consumer-to-business problem and also
served to situate the topic clearly in the realm of cutting-edge computer science research. The business of eCommerce is the business of the future, and the foundation of eCommerce will be stretchable COA based systems.

One of the most exciting aspects of scientific research is that it exposes at least as many questions as it answers. This section briefly discusses a number of open and interesting topics that beg further examination.

Analysis of the new software quality measure called stretchability led to the development of software specification axioms (SSAs) and requirements patterns (RPs). Further research to define additional SSAs and RPs, and to apply the definitions to system specification, may yield excellent results in the field of software engineering. One particularly interesting direction relates to the creation of project metric databases. Such databases would contain historical information regarding time schedules, developer experience, reusable artifacts, and programming language choice, as well as a multitude of other project management related information. This historical information could be analyzed against the RPs of each project. The resultant statistics would likely improve project-planning capabilities, including such details as budget, time, tool selection, and hiring decisions.

The perspective domain graphs (PDGs) introduced in Chapter 5 were eventually converted into a textual representation called the content description language schema (CDL-Schema). These schemas provided the basic structure for the content description language (CDL) used in content-biased communication. Two different question forms were presented in Chapter 7. In particular, one form was based on CDL with question marks used to designate the desired details, while the other was based on a CDL-Query
language. The formal details of CDL-Query, including the expression of “order by” or “group by” clauses warrants further investigation.

Another interesting avenue of research based on CDL-Schema concerns its similarities to standard relational schemas. One result of these similarities is that it should be possible to map any relational schema into a corresponding PDG. This ability enables the addition of semantic information to relational databases and also provides a mechanism for the implementation of the SHOT. General implementation details regarding storage and retrieval of SHOT content and mappings between different EOOTS dialects are worthy research candidates. Furthermore, the expression of CDL-Queries using a modified relational algebra should yield interesting results.

The Open Dispatch Tables (ODTs) and Event-Driven Open Multiple Predicate Dispatch (ED-OMPD) techniques introduced in Chapter 7 pose unique implementation challenges. One particularly interesting aspect of this research delves into programming language design to support asynchronous zero-signature method calls at all levels of development. As suggested previously, one possible approach would be to use a technique similar to the structured exception handling (SEH) provided by C++. Specifically, when the handler attempts to access a “method parameter” that was not sent by the caller, an exception could be thrown. When the information is later obtained, an EXCEPTION_CONTINUE_EXECUTION would occur. At the implementation level, this may done by storing all program variables in a form of shared memory, using getters/setters for all parameter access, and maintaining two threads, one for implementing the handler and another for assuring the handler has the data it requires.
Finally, the relationship between standard forms and PDGs warrants further study. A form, whether a paper form or an HTML form, has a hierarchical structure and typically terminates with a set of atomic values cells. In other words, a form is essentially equivalent to a granular partition. It provides a window onto the world at some chosen level of granularity. Since forms are still the preferred input method for human-computer interfaces, it stands to reason that automating form-to-PDG conversions and providing form-to-PDG mappings would greatly enhance the usability of the global system. One immediate application of this technology would be automated conversion of a PDG request into a form on a human’s personal digital assistant (PDA). This would truly be the ultimate implementation of the ancillary GUI agent.
LIST OF REFERENCES


[113] Smith, B., “Objects and their Environments: From Aristotle to Ecological Ontology,” Department of Philosophy, Center for Cognitive Science and National Center for Geographic Information and Analysis, University at Buffalo. <URL: http://ontology.buffalo.edu/smith/articles/napflion.html#N_1_>


BIOGRAPHICAL SKETCH

Joshua Greenberg received his Bachelor of Science and Master of Science degrees in computer science from Florida Atlantic University, Boca Raton, Florida. His research interests include agent technologies, software engineering, and consumer-driven electronic commerce. Joshua has worked as a software engineer for various technology companies since 1993.