DOMAIN KNOWLEDGE REPRESENTATION IN
INFORMATION SYSTEMS

by

ERAN RUBIN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES
(Business Administration)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

May 2009

© Eran Rubin, 2009
Abstract

Information Systems and software embed knowledge about the domain in which they operate. This knowledge can be very useful to various stakeholders in the organization, including developers, users, and other organizational workers. However, it is not readily accessible and usually intertwined with implementation details. Making this knowledge available would be beneficial for several reasons. In particular: 1) software often needs to be updated to reflect changes in the organization; this causes the embedded knowledge to stay current; 2) the actual system development process often incorporates the use of methods and techniques to properly record domain knowledge; 3) knowledge embedded in software is already available in a digital format; and 4) the tools typically used to manage system development (e.g. source and version controls) can be effective in management and control of knowledge. However, despite all these potential advantages, embedded knowledge is usually not readily accessible to knowledge seekers in the organization. This situation impedes the possible utilization of software-embedded knowledge.

The objective of this dissertation is to develop ways of making software-embedded domain knowledge available, accessible, and usable to organizational users. The research challenge is to identify what domain knowledge is involved in systems development, to find ways to formalize it, and to demonstrate that it can be explicitly represented in developed systems.

The research covers three main aspects: 1) identifying and formalizing embedded domain knowledge obtained in systems development processes; 2) developing methods for representing this knowledge formally to facilitate its use during and after system development and, 3) demonstrate how this knowledge can be explicitly represented in the final IS implementation code.

The first aspect, namely the nature of embedded knowledge, is addressed by analyzing the requirements engineering, systems analysis, and enterprise modeling literature in order to identify the main constructs used for domain representation. Formalization is then accomplished using ontological analysis. The feasibility of explicit representation is attained by suggesting a Model Driven Architectures (MDA) where the formalized knowledge is used to drive processing in the system.

Usability and usefulness of the ideas are demonstrated in two ways. First, case studies and examples show how domain knowledge acquired during extant methods of systems analysis can be represented using the proposed representation constructs. Second, a sample system design, supporting explicit domain knowledge representation in system code, is proposed and demonstrated via a simple prototype.
# Table of Contents

Abstract ................................................................. ii
Table of Contents ....................................................... iii
List of Tables ............................................................ v
List of Figures ............................................................. vi
List of Acronyms .......................................................... viii

1. Introduction ............................................................ viii
   1.1 Motivation ......................................................... 2
   1.2 Research Questions .............................................. 3
   1.3 Research Approach .............................................. 4
   1.4 Contributions .................................................... 6
   1.5 Dissertation Outline ........................................... 9

2. Literature Review .................................................. 12
   2.1 Background ....................................................... 12
   2.2 Knowledge Constructs in System Development ............... 15
   2.3 Knowledge in Different Steps of Systems Development .... 18
      2.3.1 General ..................................................... 18
      2.3.2 Systems Analysis ........................................... 20
      2.3.3 Requirements Engineering ............................... 21
         2.3.3.1 Data Modeling ....................................... 23
         2.3.3.2 Behavioral Modeling ................................ 25
         2.3.3.3 Enterprise Modeling \ Goal-Oriented Approaches 26
         2.3.3.4 Domain Modeling .................................... 27
      2.3.4 Semantics in Design and Implementation ............... 28
         2.3.4.1 Semantics Embedded through Software Design .... 29
         2.3.4.2 Design Patterns ....................................... 30
         2.3.4.3 Semantic Expressiveness of Programming Languages 31
         2.3.4.4 Aspect-Oriented Programming ...................... 33
         2.3.4.5 Logic Programming ................................... 34
         2.3.4.6 MDA .................................................... 37
         2.3.4.7 Requirements Engineering .......................... 38
            2.3.4.7.1 Requirements Traceability .................... 40
      2.4 Knowledge Management Systems ............................ 41

3. Domain Knowledge Constructs ..................................... 44
   3.1 Domain Knowledge in Software Development ................. 44
   3.2 Knowledge Types in Requirements Engineering ................ 46
3.3 Developing the Model of REDK ................................................................. 47
  3.3.1 Basic Model of REDK ........................................................................ 48
    3.3.1.1 Domain Knowledge in Data-Based Approaches ..................... 49
    3.3.1.2 Domain Knowledge in Behavioral Approaches ...................... 51
    3.3.1.3 Domain Knowledge in Goal-Oriented Business Enterprise Modeling
      Approaches .............................................................................. 56
    3.3.1.4 The Domain Knowledge Approach in Requirements Engineering .. 64
    3.3.1.5 Integration ............................................................................. 69
  3.3.2 Enhanced Model of REDK .................................................................. 81

4. Approach for Representing Requirements Engineering

  Domain Knowledge .............................................................................. 85
    4.1 Approach Explained ........................................................................ 85
    4.2 Enabling the Representation of REDK - Ontology ....................... 87
    4.3 Representation of REDK .................................................................. 89
      4.3.1 General Outcomes of Ontological Foundations ..................... 89
      4.3.2 Formalizing Domain Knowledge Constructs ......................... 92
    4.4 Using REDK Representation Constructs to Represent Domain
      Knowledge Elements ..................................................................... 105
      4.4.1 The Tropos Case ..................................................................... 106

5. Framework for Supporting Development with REDK 120

  5.1 General .............................................................................................. 120
  5.2 Development Framework Illustration ............................................... 120

6. Proof of Concept Description ............................................................. 140

  6.1 General .............................................................................................. 140
  6.2 Design Principles ................................................................................ 143
  6.3 KCE Definition ................................................................................... 152
  6.4 Knowledge Connection Engine (KCE) Summary ............................. 158
  6.5 Implementation ................................................................................... 159
    6.5.1 Knowledge Connection Engine Interface ................................. 159
    6.5.2 Application Subsystem Modification ....................................... 160
      6.5.2.1 Lessons Learned from Proof of Concept ......................... 167

7. Summary .............................................................................................. 170

Reference ............................................................................................. 178
List of Tables

Table 1: Requirements Engineering Domain Knowledge Constructs Mapping ............................................................................................................. 71

Table 2: Requirements Engineering Constructs in a Sample of Additional Resources ........................................................................................................ 77

Table 3: Ontological Representation of REDK Constructs .......................................................................................................................... 98

Table 4: REDK Relationships ............................................................................................................................................................................. 100

Table 5: The Tropos Case - Constructs, Meaning, and Representation .... 107

Table 6: The Tropos Case – Means End Analysis Constructs, Meaning, and Representation ........................................................................................................ 110

Table 7: Mapping of Constructs which Differ in Tropos and REDK ......112

Table 8: Additional REDK construct Representation .................................... 115

Table 9: Goals and Softgoals in Development Example .......................... 122

Table 10: System Requirements ......................................................................................................................................................... 126

Table 11: Difference Elicited in Data-Based RE ........................................ 138

Table 12: Differences Elicited in Goal-Based RE ..................................... 139
List of Figures

Figure 1: Meta model of data based modeling constructs .......................... 51
Figure 2: Meta model of behavioral modeling constructs ....................... 56
Figure 3: Meta model of goal-oriented enterprise constructs ................... 64
Figure 4: Meta model of domain knowledge approach ............................ 69
Figure 5: Meta model of requirements engineering domain knowledge constructs ..................................................................................................................... 75
Figure 6: An actor interacts with another actor to perform a service .......... 80
Figure 7: Interaction is manifested by the role construct ....................... 80
Figure 8: A resource is used by an actor in a service .............................. 81
Figure 9: Enhanced model of requirements engineering domain knowledge ..................................................................................................................... 84
Figure 10: Representation of requirements engineering domain knowledge99
Figure 11: Strategic dependency i* model of the Tropos case (Adapted from Fuxman et al. 2004) .................................................................................. 106
Figure 12: Strategic rationale i* model of the Tropos case (Adapted from Fuxman et al. 2004) .................................................................................. 111
Figure 13: REDK model of the Tropos case (Fuxman et al. 2004) .......... 116
Figure 14: Strategic-Dependancy diagram of the implementing organization domain ........................................................................................................... 123
Figure 15: Strategic-Rational diagram of the implementing organization’s domain ......................................................................................................... 127
Figure 16: Final Strategic- Rational diagram of the implementing organization’s domain ......................................................................................................... 128
Figure 17: Class diagram - data based analysis of the organization's domain ................................................................. 129

Figure 18: Data based analysis of the organization's domain refinement. 130

Figure 19: Conceptual model of Handle Request service ................................. 132

Figure 20: REDK of CRM example ......................................................................... 132

Figure 21: Part of the implementation design of the CRM System................ 137

Figure 22: The use of a knowledge konnection engine in the Framework 143

Figure 23: System design and an example instance of use................................. 147

Figure 24: Conceptual subsystem base classes. Used when representing knowledge at DKRL ................................................................. 148

Figure 25: System classes and components .......................................................... 157

Figure 26: Student class definition of original application.................................... 161

Figure 27a: New implementation of student related classes (implementation and mediator) ................................................................................................. 165

Figure 27b: New implementation of student related classes (conceptual). 166

Figure 28: Conventional development process and MDA concepts ............... 176

Figure 29: Framework development process ...................................................... 176
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOP</td>
<td>Aspect Oriented Programming</td>
</tr>
<tr>
<td>CS</td>
<td>Conceptual Subsystem</td>
</tr>
<tr>
<td>DK</td>
<td>Domain Knowledge</td>
</tr>
<tr>
<td>DKRL</td>
<td>Domain Knowledge Representation Layer</td>
</tr>
<tr>
<td>ER</td>
<td>Entity Relationship</td>
</tr>
<tr>
<td>EER</td>
<td>Extended Entity Relationship</td>
</tr>
<tr>
<td>GBRAM</td>
<td>Goal Based Requirements Analysis Method</td>
</tr>
<tr>
<td>KAOS</td>
<td>Keep All Objectives Satisfied</td>
</tr>
<tr>
<td>KCE</td>
<td>Knowledge Connection Engine</td>
</tr>
<tr>
<td>KMS</td>
<td>Knowledge Management Systems</td>
</tr>
<tr>
<td>IB</td>
<td>Instance Base</td>
</tr>
<tr>
<td>IS</td>
<td>Information Systems</td>
</tr>
<tr>
<td>MC</td>
<td>Mediator Component</td>
</tr>
<tr>
<td>MDA</td>
<td>Model Driven Architectures</td>
</tr>
<tr>
<td>MIS</td>
<td>Management Information Systems</td>
</tr>
<tr>
<td>OOA</td>
<td>Object-Oriented Analysis</td>
</tr>
<tr>
<td>OOD</td>
<td>Object-Oriented Design</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>OOSA</td>
<td>Object-Oriented Systems Analysis</td>
</tr>
<tr>
<td>ORM</td>
<td>Object Role Modeling</td>
</tr>
<tr>
<td>PS</td>
<td>Processing Subsystem</td>
</tr>
<tr>
<td>SA&amp;D</td>
<td>Systems Analysis and Design</td>
</tr>
<tr>
<td>SD</td>
<td>Strategic Dependency</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>SR</td>
<td>Strategic Rational</td>
</tr>
<tr>
<td>SDLC</td>
<td>Systems Development Life Cycle</td>
</tr>
<tr>
<td>RE</td>
<td>Requirements Engineering</td>
</tr>
<tr>
<td>REDK</td>
<td>Requirements Engineering Domain Knowledge</td>
</tr>
</tbody>
</table>

ix
1. Introduction

Information systems' code is an output of the knowledge intensive process of software engineering (Rus and Lindvall 2002). It is generally agreed that understanding an application domain is a critical prerequisite for developing software that will operate in the domain (Iscoe 1993). However, while extensive knowledge is required to develop information systems, research on defining the different types of knowledge used, the ways different knowledge elements affect the output information system implementation, and the ways to help utilize this knowledge in other contexts, is still limited.

A significant portion of the knowledge required for the process of software engineering is gathered during the preliminary stages of the Systems Analysis and Design (SA&D) process. Specifically, during the conceptual analysis phase knowledge about the purpose of the information system and about the domain of discourse is established. This knowledge shapes the implementation of the information system, and is eventually manifested into software code.

Today, it is difficult to analyze where and how domain knowledge affects the design of an Information System. The problem originates from the lack of agreement about what "domain knowledge" really is. We posit that by articulating a scope of domain knowledge in software development, as well as defining its content, we can establish the basis for effectively analyzing its effect on information systems implementation. In particular, this can enable making this knowledge explicit in the developed system.
This thesis is concerned with identifying and defining domain knowledge obtained during requirements engineering, we term “Requirements Engineering Domain Knowledge”. Further, we claim that, with a clear definition of this domain knowledge\(^1\), and with an understanding of the way it affects information systems implementation, we can develop ways to improve development processes so that Requirements Engineering Domain Knowledge can be utilized in other contexts.

In summary, this dissertation addresses the definition of a scope for domain knowledge, the definition of its content, and the suggestion of methods for its representation. Using the results of this research, domain knowledge embedded in code can potentially be utilized in contexts other than IS development.

### 1.1 Motivation

Since the introduction of IT in organizations, organizations have gradually become increasingly dependent on it. Business rules and knowledge about the domain, which used to be possessed by individuals, are nowadays often encoded in the organization’s information systems. However, the encoded knowledge, generally referred to as “domain knowledge”, is often hard to define (Ruben 1990). Since its scope and definition is not very clear, and since it is often “hidden” in the code, this knowledge is not readily accessible as a source of knowledge. Extracting domain knowledge from code requires programming knowledge as well as a deep understanding of the software and might not be practically possible even then. Technical operation of the system as well as programming languages

\(^1\) In this section we use the general term “domain knowledge”, when, in fact, the discussion applies to the scope of “requirements engineering domain knowledge”, as defined later in our work.
and techniques need to be well understood by the individual in order to be able to extract domain knowledge from a program. Even if one possesses these qualifications, it may be that only a subset of the embedded knowledge can be extracted from code.

One way to overcome the difficulty in knowledge extraction directly from code is to refer to software documentation. Software documentation can be used to express domain knowledge in an easy to understand way. However, as software changes and evolves, this documentation is very often not updated. As a result, documentation is frequently inaccurate and cannot be trusted (Clarke et al. 1999). Nevertheless, if domain knowledge could be explicitly represented as part of the IS code, we may be able to assure accurate and updated documentation.

The objective of this research is to develop ways that will advance our ability to explicitly represent domain knowledge gathered during the development processes. Such representation of gathered domain knowledge can be used to support explicit representation in system code so that this domain knowledge will be easier to extract. In this context, systems will not only serve as providers of functionalities, but also as a reliable source of domain knowledge. To accomplish this we need to establish an understanding of (1) what domain knowledge comprises; (2) at which stages of the development process this knowledge is recognized; and (3) develop constructs that will enable effective representation of this domain knowledge.

1.2 Research Questions

The following research questions emerge from the above discussion:
1. What constitutes “domain knowledge”, as referred to in system development literature, and when within the system development process is domain knowledge primarily accumulated?

2. How can we effectively represent domain knowledge, both in general and in system code?

3. Can we demonstrate ways to support explicit domain knowledge representation in development processes and in final implementation code?

Our objective is to address these research questions by carrying out research associated with the different related disciplines, in particular: requirements engineering, system analysis and design, and software development.

1.3 Research Approach

The premise guiding the research is that currently different types of knowledge are gathered during requirements engineering, in particular, domain knowledge. However, domain knowledge is difficult to define, and is implicitly represented in a somewhat inconsistent manner in final information systems’ code and thus is not usable.

Our research helps establishing common ground for understanding and communicating about domain knowledge. Our research is also concerned with the development of new practices by addressing the fact that large quantities of domain knowledge are gathered during requirements engineering, and eventually embedded in software code. We approach our research questions by referring to previous work associated with domain understanding and its incorporation into a development processes. Our first goal is to define what domain
knowledge is and suggest methods for its representation. To understand the role of domain knowledge in information systems we refer to previous research associated with requirements engineering, domain modeling, ontology, system development processes, and software engineering techniques. The examination leads us to the conclusion that models and processes used in Requirements Engineering (RE) incorporate concepts that are essential in establishing an understanding of domain knowledge (DK) in development processes.

We continue by analyzing requirements engineering approaches in order to establish a clear understanding of the core domain knowledge constructs referred to in RE. We examine the different RE approaches and the role of DK in these approaches. In each approach we extract the core DK constructs used and create a meta-model incorporating the constructs used and their relationships. We then examine each construct in the meta-models and compare it to benchmark constructs used in the Enterprise Ontology of Uschold et al. (1998). Having found the core enterprise constructs referred to in RE, we create a meta-model of the core domain constructs used in the different RE approaches. We then further test our constructs against the literature, and show agreement and show that the domain terms used in a sample of other RE cases match our constructs.

Our next step is to seek ways to explicitly represent domain knowledge elements in software systems. The approach applied for this task is to use ontological constructs for formalizing domain knowledge elements. Ontological constructs provide a clear and unambiguous way to represent real-world elements and as such are suitable for this research task.
Finally, we demonstrate how the representation constructs can be used for explicit representation of domain knowledge gathered in the development processes. Specifically, we demonstrate the use of the constructs via a case study from the literature; another requirements elicitation example incorporating data-based RE and goal-based RE; and by demonstrating implementation infrastructure to support explicit domain knowledge representation in final IS code. In the RE cases, we illustrate how all domain knowledge can be explicitly represented using Requirements Engineering Domain Knowledge (REDK), and how REDK is capable to explicitly represent some domain knowledge which may have been only implicit or not represented when not modeling with REDK representation constructs.

For demonstrating the explicit representation of REDK in implementation, we refer to the general framework of Model Driven Architectures. We demonstrate how the proposed formalization of domain knowledge, can be used to explicitly express this knowledge in IS code. This demonstration includes the conceptualization of a general system design and the implementation of system components to support explicit representation of domain knowledge. In our implementation we use software engineering paradigms such as Aspect Oriented Programming, and Object Oriented Programming, to facilitate clarity in implementation code.

1.4 Contributions

The core contribution of this dissertation is in analyzing the contents of domain knowledge, defining its core constructs, and establishing an ontology-based representation of domain knowledge acquired in requirements engineering. We use ontology to guide us how to provide effective ways for representing domain knowledge.
With the wide range of work relating to domain knowledge that is being conducted in different fields of IS, it is crucial that a clear understanding of the meaning of this term in different areas and an effective way for domain knowledge representation and modeling be established.

The knowledge about the application domain is of interest to several types of stakeholders. These stakeholders include, in particular: expected system users, business and systems analysts, systems designers and implementers, and maintenance programmers. Domain knowledge serves as a focal point for the communication of these stakeholders about the objectives, requirements and functionalities of developed systems. In addition, other knowledge workers in an organization might be interested in the same domain knowledge.

In addition, formalizing the notion of domain knowledge in systems development can help establish a coherent framework for relating different fields that conveniently use the same term while referring to different knowledge elements.

In this context, we suggest methods for enabling preservation of requirements engineering domain knowledge throughout the development process. This has potential to enable explicit representation of domain knowledge obtained in requirements engineering even in final system code. To this end, important benefits for various stakeholders can be obtained by utilizing information systems as a source for domain knowledge.

First, the Requirements Engineering (RE) process incorporates structured methods that facilitate gaining knowledge about the domain. Such methods are not conventionally employed otherwise, making Information Systems code a potential source of knowledge which is not available in alternative resources. Requirements engineering domain
knowledge that is usually implicitly embedded in information systems could be of substantial value to knowledge seekers if presented to them in explicit forms. Such explicit representation would automatically be digitally stored and managed. Moreover, often software development is subject to well defined processes for eliciting domain knowledge. This can provide a discipline helping understand what domain knowledge includes, and defining methods for its retrieval for documentation.

Second, explicit knowledge representation can help confirm that an information system is a good representation of the domain and meets its requirements. This is of particular concern to system developers. For example, Wand and Weber (1988) claim that “for an information system to be a good representation, some essential characteristics of perceived reality must be captured in the information system” (p.214). Having domain knowledge explicitly represented can help assure that these characteristics are indeed captured.

Third, as software is changed to reflect evolving requirements, the embedded knowledge is updated as well. Any change in the information system’s functional requirements will result in code changes and often changes in code will result in different functionality of the system. Considering that any knowledge in an IS must somehow be codified, and that the IS is expected to be consistent with the business rules and the domain, a significant potential source for learning and acquiring knowledge about the domain in which the information system is deployed, emerges. Source code has the potential to help organizational workers gain knowledge and understand the business and its environment, if IS embedded knowledge would be inaccurate, organizational systems would fail to function properly. Hence, if IS function properly their embedded knowledge can be assumed to be reliable. This is in contrast to domain knowledge available in other
artifacts generated during system development (typically, various documents and models). Such artifacts are not necessarily subject to the strict updating discipline needed to maintain software that complies with the requirements.

IS code possesses other inherent properties which potentially make it a significantly effective repository of knowledge. For example, since IS code is the backbone of an information system, there are many tools which enable effective browsing and querying of its contents. These include revision tools (to track the source and time of revision), comparison tools (indicating the difference between revisions), and browsing tools (enabling investigation how and when portions of code are used in the system).

We believe that the outcome of this research can improve our understanding of the software development process, what constitutes domain knowledge, and the way by which requirements engineering processes help in gathering domain knowledge. Furthermore, this research can help to better understand the relation between an actual implementation and analysis, and suggest directions for improvements in development processes. Finally, the outcomes of this research can serve to make knowledge not otherwise usable, readily accessible to various stakeholders.

1.5 Dissertation Outline

Chapter 2: Literature Review: Analysis of the domain knowledge concept. The analysis includes the examination of the term in the context of different processes taking place during IS development. Specifically, the analysis includes examination of domain knowledge constructs in Requirements Engineering, Systems Analysis and Design, and Software Engineering. The analysis reveals that an important process in establishing
domain knowledge is requirements engineering. This leads us to examine requirements engineering as a knowledge gathering activity more deeply.

**Chapter 3: Conceptual Modeling:** Analysis of domain knowledge in requirements engineering. This analysis is done by examining different requirements engineering approaches, domain related constructs incorporated in them, their integration and definition. The outcome of the analysis is a meta-model of domain knowledge obtained in requirements engineering.

**Chapter 4: Representation:** Development of a formalization for domain knowledge that can support its representation. This formalization is established by referring to ontology and some representation approaches of similar constructs in software engineering. Following the definition of the representation constructs we provide evidence for their use during goal-oriented requirements engineering processes. Specifically, we demonstrate their use in a case study adopted from the RE literature.

**Chapter 5: Comparison:** Demonstrating the use of the proposed domain knowledge in common approaches. The purpose of this chapter is to show how domain knowledge usually gathered in common RE approaches can be represented using the DK definition of this thesis. In the scope of this section we compare the representation of domain knowledge documented in common approaches to the domain knowledge that can be documented using our representation constructs. This comparison highlights the role of DK in systems analysis.

**Chapter 6: Demonstrating How Explicit REDK Representation May be Supported in Final IS code:** This chapter addresses design and implementation issues. The chapter presents an implementation of generic system components supporting explicit
representation of domain knowledge in information systems code. The implementation demonstrates a possible system in which domain knowledge is separated from other types of knowledge in the final implementation. An important aspect of the proposed implementation is that the DK serves for more than just documentation. Rather, it defines the actual processing that needs to take place in the system. This ability is based in the ontological formalization proposed in Chapter 4.

Chapter 7: Summary: This chapter summarizes the main contributions, discusses some issues related to the cost and benefit of explicit representation of DK, and suggests some future research directions.
2. Literature Review

2.1 Background

The meaning of application design and its representation has come a long way in the short history of software engineering, and today it involves much more than problem solving. Denning et al. (1989) have developed a framework for computing. They observed that many computing activities do not involve programming, but rather “transform information: their theory, analysis, design, efficiency...”; Hence, we learn that much of software development today is about the knowledge required for analysis and design of information, rather than simply fulfilling requirements. Moreover, much of the transformed information is embedded in system code, as indicated in the same source: “access to the distinctions of any domain is given through language, and most of the destinations of computing are embodied in the programming notation” (Denning et al. 1989). However, can we clearly state what knowledge is embedded in system code, and how it is embedded?

The system development process has traditionally been considered to gradually move from the “problem space” to the “solution space” (Booch 1987). As such, system source code is traditionally considered to be defined entirely in terms of solution space. However, as different programming paradigms emerged, the boundary between the problem and solution domains has blurred (Henderson-Sellers and Edward 1990). It was recognized that when compared to the implementation phase, the design phase is more loosely defined since it consists of progressive decompositions towards more and more detail (Sommerville 1989). On top of that, the design stage is considered more of a creative one, rather than a mechanistic process (Turner 1987). This situation only intensified as time progressed.
When tracing the history of software abstraction, it is evident that purpose of programs has changed from merely a tool for refining the way computers work, to legitimate intermediates between the problem and the computer (Abbot 1987). Initially, programs did not exhibit problem level information and were very difficult to understand or modify. It was not expected that a good programmer should make a program understandable and modifiable, but rather programs were measured merely by their efficiency and whether they accurately solved a specific problem or not. As time passed, the software engineering revolution took place. Software engineering now emphasizes that programs should be written and designed so they would be understandable and maintainable. As a result, software abstraction took place in different forms. An important step in this direction was the introduction of Object Oriented Programming (OOP). OOP moved programs much closer to the problem domain. Indeed, it is claimed that the success of OOP is not necessarily due to its technical advantage, but more due to enabling programmers to think in terms of the problem domain (Abbot 1987).

The more source code related to the problem space, the more difficult it was to distinguish between analysis, design and the implementation. Different views appeared regarding the difference between the output of Object Oriented Analysis (OOA) and the output of Object Oriented Design (OOD) (e.g. Jacobson 1995, Kaindl 1999, Rumbaugh et al. 1991, Shlaer and Mellor 1992). According to Kaindl (1999), some authors (Coad and Yourdon 1991, Jacobson 1995) pointed out that OOA refers only to domain objects, while OOD refers also to future system objects. Yet other authors did not clearly distinguish between the two aspects (Rumbaugh et al. 1991, Shlaer and Mellor 1992). This state of affairs led to the prevailing perception that code is the only true design of a system. Blum
(1996) states that “A program is not an object but rather it as a design of an object”; Reeves (1992) claims that programming is about designining software and “the final source code is the real software design”. These ideas are part of the general movement towards agile development. It has often been claimed that systems evolve in response to changing needs but that the design documents fail to reflect these changes (Clarke et al. 1999, Bennett et al. 1991, Laitinen 1996). Since maintenance and update constitute the majority of a system’s lifecycle (Clarke et al. 1999), this failure has accounted for much of the criticism of traditional development processes. The traditional development process was claimed to be too cumbersome and to provide limited benefits (Fowler 2004). However, agile methods have been criticized for avoiding documentation, which is needed for knowledge sharing and for reducing knowledge loss (e.g. when team members become unavailable) (Abrahamsson et al. 2002). While traditional development methods over-document, a weakness of agile methods is not generating sufficient documentation (Paetch et al. 2003).

Basically, while the arguments presented above have merit, it currently cannot be claimed that the source code and other design models created during the development process are equivalent. Two major differences exist between the two:

1. The source code is not as accessible to different stakeholders in the organization as design documents are. Basically, only programmers can understand source code. Moreover, familiarity with the development environment and the programming language syntax is needed in order to navigate through source code.

2. The source code contains much more information than included in the intermediate design documents. While the code embeds much of the information available in design documents, it also incorporates implementation-related details. Examples are handles to
system objects (e.g. files and devices), algorithmic details (e.g. how an algorithm is accelerated for performance reasons), networking details (e.g. use of ports and socket information), and user interface details (e.g. the pixel location of a specific button on a screen).

The current state of affairs is that software is assumed to represent reality on the one hand, while dealing with physical issues such as memory management and efficiency, on the other hand. This issue brings together conflicting ideas and entails a deficiency by its very nature. Design models and the system source code are highly related and both incorporate domain knowledge. However, fundamental properties, associated with the structure and composition of code, render source code and design documents two distinct artefacts. This situation impedes the use of code as an explicit source for the knowledge embedded in it.

We therefore examine different types of knowledge used and modeled at different stages of system development. This analysis can help establish a better understanding of stages of the development process where portions of reality are understood and how they are represented.

2.2 Knowledge Constructs in System Development

In their work Mylopoulos et al. (1993) identify the different worlds related to the information system: the subject world (application domain), the usage world (user models, environments), the system world (software requirements, design), and the development world (teams, methodologies). The different worlds are classified as follows:
• The subject world is the domain about which the proposed information system is to provide information.

• The system world includes specifications at different levels of implementation detail on what the information system does. It may include functional requirements, and actual implementations of the system. The correspondences between the information maintained by the information system and the intended subject world, is part of the system world.

• The usage world describes the environment within which the system is embedded. Such descriptions often take the form of input/output relationships, but may also include the different classes of end users of the intended system or the kinds of interfaces supported by the system. The usage world may also include descriptions of the organizational environment within which the system will function, including office procedures.

• The development world focuses on the entities and activities, which arise as part of the design process itself. These would include the composition of the design team, responsibilities of each team member, design decisions, development tools, supporting documentation, etc.

Parnas and Madey (1995) propose different classes of “knowledge documents”, which serve to “separate concerns”. The documents are defined according to their content rather than their form. They describe the different documents as follows:

• System Requirements Document: A document that treats the complete computer system as a "black box". The document includes a description of the environment that identifies a set of parameters of concern to the system’s users. The document
also describes the relationships between the values of these parameters that result from different constraints.

- System Design Document: A document that identifies the components within the system and describes how they communicate, and what the input/output parameters are.

- Software Behavior document: This document records design decisions and provides description of the actual behavior of the software. Specifically, they claim this document describes the knowledge identified in the system requirements and the design document that is materialized in the software.

- Software Module Guide: This is an informal document that describes the division of the software into modules and states responsibilities of each module. Each module is associated with a Module Interface Specification, which identifies functions that can be invoked from outside the module.

- Uses-relation Document: This document defines interrelations between modules. The document describes the modules used by other modules.

- Module Internal Design Document: This document describes the module’s data structure and its interpretation

- Service Specification and Protocol Design Document: This document describes communication protocols between components of the system

- Chip Behavior Specification: This document includes a black-box description of the behavior of hardware chips.

With these classifications we notice the “worlds” which incorporate domain knowledge, and those that are concerned with the developed information system.
Specifically, the subject world, usage world, and parts of the system world, incorporate domain knowledge. However, it can be seen that most of the documents created in those worlds include a description of the system and its implementation process. The document which incorporates large quantities of domain knowledge appears to be the System Requirements Document. It therefore appears that requirements are highly related to domain knowledge. However domain knowledge is also found in the system itself and in the interaction between the system and the environment.

With the broad understanding of domain knowledge elements, in the next few sections we further review the knowledge incorporated in the different documents created during the development processes, how it is transferred and how it is represented.

2.3 Knowledge in Different Steps of Systems Development

2.3.1 General

Information Systems (IS) can be viewed as representations of the domain in which they operate (Wand and Wang 1993). Related to this, Avison and Fitzgerald (1995) state that an information system is required to help the organization in analyzing its business, along with its environment. Since IS help control, manage, and improve efficiency of processes taking place at the organizational domain, both the design of an information system and the information system itself require a clear understanding and accurate representation of the domain of discourse. Indeed, the initial phase in system development, systems analysis, is largely concerned with domain understanding.

The description of the processes of gaining an understanding of how an information system (IS) can support business needs, as well as how the IS is designed, built, and
delivering to its users, is traditionally based on an underlying model commonly referred to as the “System Development Life Cycle” (SDLC). A SA&D methodology can be defined as a method to implement a SDLC aimed to implement an information system (Dennis and Wixom 2003).

Different methodologies assume different phases in SDLC. However the SDLC is generally based on three core phases: analysis, design, and implementation (Henderson and Edwards 1990). In the analysis phase, reasons for the need for an information system are understood and sequentially converted to user needs analysis and feasibility study (Davis 1988). In the design phase different levels of concern are addressed, and logical and physical designs are produced. Finally, in the implementation phase, a computer program is written, tested, validated, deployed and maintained in the organization.

Undoubtedly, the analysis phase is tightly related to organizational domain knowledge. Due to the tight relation between and IS and its domain, every IS project begins by understanding the real-world domain represented by the IS. One of the initial tasks in the analysis phase is concerned with describing the real world domain through conceptual models. These models are descriptions of the real-world independent of any information systems or information technology aspect.

Conceptual models are fundamental at the analysis phase and consequentially serve as a basis for the design phase. Their purpose is both to serve as communication medium for understanding of the domain, and to serve as a guide for IS design (Kung and Solvberg, 1986). Conceptual models represent the problem understanding (Devedzik 2002), and facilitate understanding among various stakeholders in the organization (Mylopoulos and Borgida 1997). “Conceptual modeling is the activity of formally describing some aspects of
the physical and social world around us for purposes of understanding and communication” (Mylopoulos 1992).

During the design phase, implementation concerns, or physical concerns, are considered for the first time and are added to the conceptual model (Juristo and Moreno 2000). However, the ways for preserving conceptual ideas during design and final system code have been largely overlooked in IS research. In the following sections we conduct an analysis of semantic representation in systems analysis and design. We look into different processes taking place during the analysis phase, and we examine some methods and techniques associated with semantic representation used during the design phase.

2.3.2 Systems Analysis

Traditionally, systems analysis was considered to be the phase in which reasons for the need of an information system are understood and subsequently converted through to user needs analysis and feasibility studies (Davis 1988). Realizing that inaccurate requirements account for many problems encountered during software development (e.g. Alford 1979, Nakajo and Kume 1991) the concept of systems analysis has undergone some transformations. Gradually reference to the systems analysis phase has shifted from “Systems Analysis” to “System Requirements Analysis” (e.g. Grady 2006), “Requirements Analysis” (e.g. Hay 2003), and most commonly today to “Requirements Engineering” (RE). There is no clear distinction between the different terms. However, it appears that in earlier days Systems Analysis referred to well defined processes leading to coherent development, and RE refers to methods for understanding and capturing organizational needs. While conceptual modeling has always been an important component of systems
analysis and design, the process still puts much emphasis on the operation of the new system. On the other hand, RE emphasizes techniques and approaches stressing the understanding of the context of, and reasons for, requirements. As stated by Loucopoulos and Kavakli (1999), traditional systems analysis has proved to be too monolithic and lacking facilities for dealing with highly complex, multidimensional, and distributed systems. In the traditional paradigm, little attempt is made to understand how the proposed system will effect the enterprise itself.

2.3.3 Requirements Engineering

Requirements specification has a multifaceted role. For example, it can and should be the means by which stakeholders and systems analysts communicate. The specification itself can be used for clarifying a situation in which the intended system will operate, or about its environment i.e. the organizational context (Loucopoulos and Kavakli 1995). Engagement in an RE process presupposes that some new computer based system could be useful, but such a system will change the activities that it supports. Therefore, RE needs to be sensitive to the way people perceive and understand the world around them, the way they interact, and how the environment of the work system affects their actions. RE is concerned with interpreting and understanding stakeholder terminology, concepts, viewpoints and goals (Nuseibeh and Easterbrook 2000). Therefore, RE research and practice draws on the cognitive and social sciences to provide both theoretical grounding and practical techniques for eliciting and modeling requirements.

Four broad tasks are identified as constituting the RE process (Nuseibeh and Easterbrook 2000): Requirements elicitation - understanding the organizational situation;
Requirements negotiation – establishing an agreement on the requirements of the system; Requirements specification – defining requirements and Requirements validation – ensuring that the derived specification corresponds to the original stakeholder needs.

From these four tasks, requirements elicitation is the process that most profoundly depends on domain understanding. Elicitation is the activity considered to be the first step in the RE process. This activity usually involves creating different types of models. As stated by Nuseibeh and Easterbrook (2000), to a great extent the elicitation technique used during RE is driven by the choice of modeling scheme, and vice versa. To this end, Nuseibeh and Easterbrook (2000) mention four general categories of modeling approaches used in RE: data modeling, behavioral modeling, enterprise modeling, and domain modeling. It is important to note that the approaches are not mutually exclusive. Rather, each approach takes a different view as to the aspect that should be emphasized to facilitate requirements elicitation.

Data modeling approaches deal with understanding the information that needs to be stored, maintained, and managed by the information system. Behavioral modeling approaches deal with the dynamic or functional behavior of stakeholders and systems, both existing and required. Enterprise modeling approaches deal with understanding an organization’s structure, the business rules that affect its operation, and the goals, tasks and responsibilities of its members. Finally, domain modeling approaches deal understanding the impact of the system on the domain, and the way it helps control the domain.
2.3.3.1 Data Modeling

Data modeling approaches originate from the more traditional systems analysis and design methodologies and aims to handle issues associated with the operation of the system. Since information systems use and generate large volumes of information, the information needs to be understood, manipulated and managed. Data modeling aims to help stakeholders make decisions about the information that the system should represent, and the correspondence between the information system and the real world (Nuseibeh and Easterbrook 2000).

Data modeling approaches underlie the first RE processes. These processes follow the SADT proposal (Ross 1977). Greenspan et al. (1986) developed RML (Requirements Modeling Language), which formalized SADT by using ideas from semantic networks and semantic data models. The result was a requirements modeling language in which entity and activity classes were organized into generalization hierarchies (Mylopoulos et al. 1999). Hence, both Entity-Relationship (ER) modeling and Object-Oriented modeling are used in this approach.

ER modeling was proposed by Chen in 1976 for the specification of logical database structures. Since then it has been used often in the database community as a first step in the specification of databases. ER diagrams essentially consist of Entities, Relationships and Attributes. Various extensions (e.g. Ozkarahan 1990) to the ER model have been proposed yielding the Extended Entity Relationship model (EER), which also incorporates generalization, specializations relationships.

The first Object-Oriented Analysis (OOA) techniques were proposed about two decades ago. The Object-Oriented Systems Analysis (OOSA) technique (Shlaer and Mellor
1988) adopts the Entity-Relationship (ER) model to capture the declarative aspects of a software system. This was soon followed by the Object-Oriented Analysis (Coad and Yourdon 1990) and the Object-Oriented Modeling Technique (OMT) (Rumbaugh et al. 1991).

According to Davis (1993), the Object Oriented Analysis technique consists of five activities: specifying objects, attributes, structures, services and subjects. According to Coad and Yourdon (1990), objects are “an encapsulation of attributes and exclusive services; an abstraction of something in the problem space, with some number of occurrences in the problem space”. Similar to ERM, the object-oriented techniques include objects, attributes and structural relationships – Generalization (termed gen-spec in OOSA) and Composition (Whole-part in OOSA). A significant difference from the EER approach is that in the Object Oriented paradigm objects additionally incorporate services/methods.

The great appeal of OOA is that the whole software development process can be streamlined and simplified by having the same building blocks (objects, classes, inheritance, etc.) used in all phases of development, from requirements to implementation.

Indeed, OOA evolved from Object Oriented Design (Booch 1991). At design time, an object is generally considered to be an abstract data type – i.e. an encapsulation of data along with functions. With OOA transition from requirements to design is supposed to be more straightforward. However the disadvantage is that the criteria for object selection at requirements time may be different than for design time. The criteria at requirements time must include some notion of a real world entity that is important to understanding the problem. At design time this is not essential.
In order to exploit the potential of object orientation in streamlining development, UML emerged as unified modeling language incorporating all elements facilitating Object Oriented modeling – from design to implementation. However, this notion has brought analysis even closer to implementation and since the data modeling view is oriented towards the data managed by the system, UML models will typically not be good representations of the real world domain (Evermann and Wand 2005).

2.3.3.2 Behavioral Modeling

Behavioral modeling approaches concentrate on the dynamic or functional behavior of stakeholders and systems, both existing and required. However, in behavioral modeling the distinction between modeling the existing system, and modeling the future system is frequently blurred by the use of the same modeling techniques for both (Nuseibeh and Easterbrook 2000). Similarly to data modeling approaches, behavioral modeling approaches tend to concentrate on examining the behavior of the system to be built more than the current system or its environment. However, unlike data modeling approaches, behavioral approaches emphasize the changes that take place. These are often associated with the introduction of the information system to facilitate changes. To this end, since at the early stage, details of the operation of the system are not essential, behavioral modeling approaches would typically be applied in subsequent phases of the requirements engineering process, such as requirements refinement. In these phases, emphasis is naturally put in the future system rather than the current one. That is, emphasizes on how changes are accomplished and under what circumstances they take place. Research on the
behavioral aspect of software systems includes formal notations, such as state charts (Harel 1986) and Petri Nets (Petterson 1977).

2.3.3.3 Enterprise Modeling \ Goal-Oriented Approaches

The Enterprise Modeling approach, also referred to as Goal-Oriented, emerged in the mid 80s and some of its examples include i* (Yu 1997); GBRAM (Anton 1996), KAOS (Dardenne and van Lamsweerde 1993), and EKD (Kavakli and Loucopoulos 1999).

Enterprise models aim at expressing organizational objectives or goals and associated tasks and resources (Yu 1997) and are about describing, in some formal way, a social system with its agents, work roles, goals, responsibilities, and the like. Enterprise modeling deviates from traditional requirements engineering as it takes a step away from the view of requirements specification, in which a functional specification stresses the definition of the desired service of the intended system. As such, enterprise models are intended to free the RE process from over-reliance on functional specifications. The use of functional specification for requirements analysis has been criticized as having a number of undesirable side effects (Loucopoulos & Kavakli 1995). In particular, such use can generate a large amount of excess detail, with which both requirements holders and requirements analysts have to deal. As well, when functional specifications become the focal point of requirements analysis, decisions related to system boundaries might be made before any understanding is gained of the real needs of the requirements holders (McDermid 1994). In other words, emphasis on functional specifications tends to deflect attention from important aspects. These aspects include the objectives of the system itself
and the relationship of these to organizational objectives, and other desirable properties of the system.

To overcome these deficiencies, enterprise modeling provides the means for describing the current structure of the enterprise, its missions and objectives. Therefore, goal modeling is often emphasized in this approach, and today the approach is often referred to as “goal-oriented” rather than enterprise modeling. According to this view, since high-level business goals can be refined repeatedly as part of the elicitation process, goals can lead to requirements that can then be operationalised (Dardenne and Lamsweerde 1993).

2.3.3.4 Domain Modeling

Domain modeling approaches aim to provide an abstract description of the world in which an envisioned system will operate. Moreover, according to this approach there is a very close association between domain knowledge and requirements engineering. This approach is not associated with any particular method or tool. Rather, it emphasizes the importance of certain information that is often absent or implicit. One of the problems, or “dark corners” (Zave and Jackson 1997) of RE that motivate this approach is that there is no clear distinction between the actions controlled by the environment and those controlled by the machine. Hence domain modeling RE is highly related to the work done in this dissertation. Domain modeling approaches emphasize the need to explicitly identify elements controlled by the environment and elements controlled by the machine. This dissertation aims at identifying the portions of the environment that are represented when using RE methods of different kinds. Earlier views of this approach were provided by Fischer (1994). A more recent application of this approach is given in problem frames (Jackson 1995, Hall et al. 2005)
2.3.4 Semantics in Design and Implementation

In this section we discuss the conventional perspective of representing semantics in subsequent phases of the development process – in design and implementation.

Interestingly, most work done about representing domain semantics in information systems was primarily associated with data modeling. A likely reason may be that accurate semantic representation of domain data covers a large portion of the domain semantics. Data deal only with the static aspects of a system, and their understanding can improve system understanding significantly. We find that the methods for conveying domain semantics of dynamic aspects are not as widely addressed as methods for conveying static aspects of domain semantics in data models. This state of affairs is evident when examining areas of research within software engineering associated with domain representation. While the different database models emphasizing the ability to express more semantic information are classified in one area of “semantic data models”, the literature associated with models and methods that convey better representation of domain information in system code uses diverse terminology and is rarely discussed as supporting domain semantic information. When examining the literature, one can find terms such as “knowledge abstraction”, “aspect programming”, “design patterns” and more; all of which are discussed in isolation, without being grouped together under any classification scheme. Most probably the reason lies in the fact that each of the above paradigms emphasizes issues other than domain semantic representation. Some emphasize changeability (e.g. Aspect Oriented Programming), others emphasize development correctness (e.g. Design Patterns), yet other
emphasize flexibility and reuse (e.g. Abstraction). In this section we examine the different approaches under the perspective of domain semantic representation.

2.3.4.1 Semantics Embedded through Software Design

In order to arrive at a certain system design, knowledge about the domain must be possessed by the programmer. This knowledge may be explicitly or implicitly coded in the software program. An example of some explicit semantic information in a system is found in Object Oriented Design. Chidamber and Kemerer (1991), suggest that a program design can be notated as $P=(A,R_1...R_n,O_1...O_n)$, where $A$ is a set of object elements, $R_1...R_n$ are empirical relations on objects, and $O_1...O_n$ are binary operations that can be performed on the objects. Booch (1986) defines object-oriented design to be the process of identifying objects and their attributes, identifying operations required by each object and establishing interfaces between them. It follows that, when $A$, $R$, $O$ relate to phenomena in the domain, domain semantics is represented. This issue is also related to data modeling and conceptual modeling (Wand 1989).

Extended semantics in enhanced object-oriented approaches can be found as well. For example, in Role-Based analysis (Reenskaug 1995), roles are incorporated in the model. Similar to a class, a role is a description of a set of objects. While the class describes the set of objects that exhibit common characteristics, roles reflect relationships to other classes. With roles, one can describe the attributes that must be exhibited by all objects playing the roles and the sets of messages a role may send to another role. In role based analysis, types of objects change according to their relationships with other objects (Drossopoulou et al. 2001). Roles provide information about how the actions of the system interact with object and data structures. In any case, in the design of classes, the use of a specific set of types
of objects, their attributes and their communication protocols reflect information on domain semantics. Hence, by the very design of an OO program, domain semantic information is included in the system code. In fact, work has been done with the purpose of automatically extracting objects when the programming language is not object-oriented. Siff and Reps (1999) used concept analysis to identify modules in legacy code. In this work, the authors examine sets of different objects, attributes and their relations (where the three define contexts). However, as suggested in their article, “modularization reflects a design decision that is inherently subjective, and it is unlikely that the modularization process can ever be fully automated”. The design of a program conveys much information that cannot be gathered automatically. Two programs may be functionally equivalent; however one may convey more semantic information by design than the other.

2.3.4.2 Design Patterns

In software engineering the contexts of certain semantics can be reflected by elements called Design Patterns (Gamma et al. 1995). Design Patterns are static design elements specified in programs. A design pattern indicates a solution to a problem in a context. It is claimed that by using design patterns the semantics of the program, the context of the problem domain, and the way it is handled, can be more easily understood. In general, a pattern has four essential elements (Gamma et al. 1995): the pattern name, the problem description, the solution, and the consequences. Hence, by using specific design patterns, the semantics of the problem domain, which may include relationships between objects, consequences of using them, and some of their scope, are conveyed in the program. It should be noted, however, that design patterns do not necessary capture semantics about the environment. Rather they may capture semantics about the design, which may relate to
physical aspects of the system. Design patterns identify the participating classes and instances, their roles and collaborations and the distribution of responsibilities (Gamma et al. 1995). A noticeable advantage of design pattern is in their ability to convey much semantic information about the domain. The study of Cline (1996) states that one of the benefits of design patterns is that “many of the design philosophies are captured explicitly rather than strictly being in the head of the original development team”. If we accept the fact that a design philosophy reflects how domain semantics are represented in code, having design philosophies explicitly stated in code serves as a tool to help understand how domain semantics are represented. In fact, it may very well be that the implantation does not make use of all the constraints and information embedded in the pattern, however using a specific pattern can convey valuable semantic information in the program, about the environment, the system, and about the system design.

2.3.4.3 Semantic Expressiveness of Programming Languages

Programming languages are specified by rules for forming correct statements, organizing them into modules, and most importantly in our context, semantics of the statements (Appleby and Vandekopple 1997). I.e., they describe methods to express the problem a program is to solve, and therefore, also express the domain of discourse. Programming languages can be quite different at both the conceptual level and the implementation level. A language is organized around a particular conceptual model (Appleby and Vandekopple 1997). The semantic definition of a programming language describes the effect of executing or evaluating any syntactically correct expression. Interestingly, while there is a widely accepted method for describing the syntax of a
programming language, namely using BNF (Backus-Naur Form), there is no widely excepted method for expressing the semantics of a programming language (Zhang 2004).

Two major paradigms in programming languages are the imperative and declarative paradigms. In imperative languages, the semantics of a programming language are associated with defining state changes in the software artefact. Examples of imperative paradigm programming languages are the object based paradigm and the block-structured paradigm. In declarative languages, the semantics of the programming language are associated with specifying relation or function (Appleby and Vandekopple 1997). The literature view is that relations and functions are at a higher level than properties and states, and therefore enable higher level semantics then declarative languages do. Examples of declarative programming paradigms are the Logic programming paradigm, the Functional paradigm, and the Database Language paradigm. The paradigm probably most intimately associated with semantic expressiveness is Logic Programming. In the broadest terms, Logic Programming enables expressing axioms and rules by which new facts can be generated. However logic programming can not easily express functional semantics. Purely functional languages enable expressing only functional issues in the domain. There are no assignments or parameters in those languages. LISP is an example of such a language. While assignment is possible, it is not a key element in the language, but rather is viewed as an added functionality supported by the language. The use of lists as a metaphor to everything the program can do is the fundamental element of LISP. The basic idea in database languages is that they enable semantics associated with facts that continue to exist after a computer program terminates. Thus, they support semantics associated with persistence and management of change.
2.3.4.4 Aspect Oriented Programming

Aspect-oriented programming enables the separation and encapsulation of concerns that crosscut modules. An example of such a concern is logging functionality. If the system requires logging of every transaction made, this aspect relates to all objects and modules in the system. Hence it is an aspect, which can not be mapped to a particular object. Aspect-oriented programming is a programming paradigm that takes another step towards capturing cleanly more kinds of design concerns within source code (Kiczales et al. 1997). This programming paradigm provides explicit language support for modularizing design decisions that crosscut modules in a program. Instead of spreading the code related to a design decision throughout a program’s source, a developer is able to express the decision within a separate, coherent piece of code (Walker et al. 1999).

In effect, by using AOP, programmers can better differentiate the types of “concerns” in program code. For example, in the logging case, if logging is required only for debugging purposes, AOP enables programmers to separate testing based information from requirements based information and thus it provides a “cleaner” and more effective way to design programs.

AOP is becoming increasingly popular and is implemented in a number of programming languages; probably the prevailing one is AspectJ (Kiczales 2001). Perhaps one of the most persuasive indicators that AOP is becoming a de-facto paradigm is the fact that Microsoft started supporting some of the ideas of AOP as part of its .NET framework. AOP has great appeal in the software engineering field, and there is much evidence for its effectiveness in software maintenance processes. For example, Walker et al. (1999) describe an empirical study which compared the use of AspectJ with simple Java. The
study showed that AspectJ results in more efficient change to programs. In a more commercial setting, Papapetrou and Papadopoulos (2004) demonstrated the favorability of AspectJ.

2.3.4.5 Logic Programming

Logic programming is predominantly used in AI to model human behavior, reasoning, learning, planning, and action taking. Perhaps this state of affairs further led researchers to define knowledge in programs in a similar manner. Specifically, Newell (1982) introduced the Knowledge Level as a new systems level. The system is an agent, components at the knowledge level are goals, actions and bodies, and the medium is knowledge. The medium of behavior is the principle of rationality—namely, actions are selected to attain the agent's goals.

Logic programming is a declarative programming paradigm based on the idea that the language of first-order logic is well-suited for both representing data and describing desired outputs (Kowalski 1974).

A pure logic program consists of a set of rules, also called definite Horn clauses. Each such rule has the form head body, where head is a logical atom and body is a conjunction of logical atoms (E.g. Drives Car <- Driver and HasCar). The logical semantics of such a rule is given by the implication body implies head. Basically, classical logical systems are deductive systems. Namely, given a database Δ and formula Q, we ask whether Δ can prove Q. Classical logical systems can be classified according to the form Δ and Q can take, and what kinds of consequence relations and algorithmic systems are involved (Gabbay 1994). According to the same source, in practical systems other operations are of importance such as updating, abduction, and explanation. These are the mechanisms which
accompany the deductive component and which further help categorize logic based systems.

Central to traditional logic is the notion of logical consequence, which states that a sentence is a consequence of a set of premises—if all the sentences in the set were true, then the conclusion would have to be true too (Tarski 1956). Formally, a sentence \( \alpha \) is a logical consequence of a set of sentences \( A \) means that a model (an interpretation in which the semantic meaning of all sentences in \( A \) is true) of \( A \) is also a model of \( \alpha \). Or, in other words \( A \) entails \( \alpha \).

Another basic principle is resolution (Robinson 1965), which in essence links semantic operations to syntactic meanings. Resolution is a process including logical rules, which is a complete and consistent logical system. That is, proving facts using a process of resolution will yield semantically valid conclusions.

An important aspect of logic programming is that it is declarative in nature. It facilitates the writing of declarations, or truths. Declarative languages are considered to be higher level than imperative languages, because the programmer works with concepts rather than storage location of the machine itself (Appleby and Vandekopple 1997).

Probably the best known outcome of Logic Programming is the programming language named PROLOG (Colmerauer 1972). However, it is important to understand that PROLOG is only one of the possible implementations of the logic programming concept, as it was described here. PROLOG, however, demonstrates how domain semantics can be represented in program code. PROLOG is described as descriptive (Genesereth and Ginsberg 1985), and declarative. I.e. a programmer declares what is the problem to be solved (e.g. Sort([1,3,5,2],Answer)), and describes the domain of disclosure in the program.
In the database context, the query language Datalog was designed and intensively studied (See, e.g. Jaffar and Maher 1994).

Many different methods for enhancing the semantics of logic systems have been proposed in the literature. Generally, a large variety of domains has been studied in the context of logic programming, many of which are concerned with human behavior. Indeed, from the very beginnings of research in AI, logic has played a central role. Researchers observed that AI programs must be based on large quantities of knowledge that must be somehow represented in the program. This knowledge is common-sense knowledge about human environment and behavior. Logic is often referred to as related to AI like mathematics is related to science. That is, logic is often seen as the basic tool to enable creation and testing of AI applications (Gabbay 1993). One of the most important roles of logic in AI is as a source of languages and logics for artificial reasonless- processes of constructing arguments for adopting some cognitive attitude toward a given proposition. Logical reasoning is thus used to implement psychological processes of human resonating in artificial ways.

Using logic is dominant also in representing human planning, actions and consequences. In a classical paper, (McCarthy 1959), discusses the problem of planning the way to the airport, in which an initial state, described by the formula \( at(I,desk) \), needs to be turned into a goal state that satisfies \( at(I,airport) \). This work has been extended to representing other issues associated with action and planning. For example, representing causal dependencies that do not involve actions, where the cause and effect may not even be separated by time interval. For instance, if an individual (\( Ind \)) has a watch on his/her wrist, than \( at(Ind, airport) \) is a cause for \( at(watch, airport) \) (Lifschitz 1999). Other work
on this topic involves the execution of actions in an incomplete state - a state not fully described and for which answers to queries can be unknown as opposed to pure 'true', 'false'. This issue was addressed in the studies of Gelfond (1991, 1993).

Since the early nineties, Logic Programming has been intensively used for studying another form of human behavior- learning. At around this time, the term of inductive logic programming was first introduced by Muggleton (1991). This is the study of inductive machine learning within the representations offered by computational logic. Inductive learning techniques generalize specific observations into general laws, which can be used for explanatory or predictive purposes. Machine learning is associated with predictive induction. Data mining techniques have given popularity to descriptive induction (explanatory purposes). Examples of research where inductive logic programming is used can be found in biochemistry, as done in Bratko and Muggleton (1995).

2.3.4.6 MDA

The Model Driven Architecture (MDA) is a framework for software development defined by the Object Management Group (OMG). This framework is related to our work as it aims at facilitating structural transformations between different design phases.

Key to MDA is the importance of models in the software development process. Within MDA the software development process is driven by the activity of modeling the software system (Kleppe et al. 2003).

The MDA development life cycle is not very different from the traditional life cycle. The core difference between the two is that the artifacts created in MDA are formal models, i.e., models that can be understood by computers (Kleppe et al. 2003). However, there is no such restriction in the traditional development cycle. Central to MDA is the
notion of creating different models at different levels of abstraction and then linking them together to form an implementation. Some of these models will exist independent of software platforms, while others will be specific to particular platforms. Each model is expressed using a combination of text and multiple complementary and interrelated diagrams (Mellor et al. 2004).

A mapping between models is assumed to take one or more models as its input and produce one output model. The rules for the mapping are described by mapping rules within a mapping function; the mapping is an application of the mapping function. These rules are described at the meta model level in such a way that they are applicable to all sets of source models that conform to the given meta model.

The Model Driven Architecture emerges as a good starting point to help frame our work on knowledge transformation in a more general development framework.

2.3.4.7 Requirements Engineering

One of the first definitions of requirements engineering was given by Ross and Schoman (1977), who stated that "requirements definition is a careful assessment of the needs that a system is to fulfill. It must say why a system is needed, based on current or foreseen conditions, which may be internal operations or an external market. It must say what system features will serve and satisfy this context. And it must say how the system is to be constructed". Thus, the RE process helps identify why a system is needed, its functionalities, and the constraints on providing these functionalities (Lamsweerde 2000). This definition shows how complex and broad the process of requirements engineering is. In the process of requirements engineering different frames of reference can be recognized in the process of analyzing and modeling system requirements:
1. One frame of reference is the environment in which the system is to operate. This frame of reference is used for analyzing the contextual goals of why a software system is needed.

2. Another frame of reference is the system itself, which is used for analyzing how the system should be constructed.

3. In between, there is a frame of reference which “mixes” the two for addressing the question of what system features will serve to satisfy the need for the system identified in frame of reference 1.

Modeling has an important role in requirements engineering. The existing system and alternative hypothetical systems are molded to serve as common grounds to the different activities described above. Therefore, most of the research in requirements engineering has been devoted to techniques for modeling and specification (Lamsweerde 2000). The output of the requirements engineering process, materialized as requirements specifications, is generally categorized to functional requirements and non-functional requirements. Functional requirements refer to how the system will work. That is, these requirements include algorithms, data structures, input output settings, etc. The functional requirements, however, are complemented with non-functional requirements. These requirements impose constraints not associated with functionality. For example, such constraints may relate to performance and security. Alternatively, it can be claimed that non-functional requirements help describe the non behavioral aspects of functional requirements (Wiegers 2003).
2.3.4.7.1 Requirements Traceability

Since this dissertation is concerned with modeling and representing domain knowledge, an area somewhat related to this research is that of Requirements Traceability (Gotel and Finkelstein 1994).

As indicated in this dissertation, eliciting requirements incorporates the use of domain knowledge as input. Related to this, tracing this input is related to requirements traceability. However, requirements traceability and the work of this dissertation are clearly in different domains of concern. Our work aims at preserving domain knowledge and using it in other contexts, beyond the specific project requirements. Our work does not look into relating specific documents or code segments to a specific set of requirements. Rather we look at ways to facilitate the use of domain knowledge obtained in requirements engineering. Requirements traceability, on the other hand, concentrates at the requirements specification process, focusing on how changes in inputs affect a project's requirements, and visa versa.

According to ANSI definition, "A software requirements specification is traceable if (i) the origin of each of its requirements is clear and if (ii) it facilitates the referencing of each requirement in future development or enhancement documentation" (ANSIAEEE Standard 830-1984). Using this definition, Gotel and Finkelstein (1994) defined requirements traceability as follows:

"Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)" (Gotel and Finkelstein 1994, p.97). That is, a distinction is made between pre-RS requirements traceability and
post-RS requirements traceability. The former refers to the traceability of requirements from statements of diverse sources to their integration into requirement specifications, and visa versa. The latter refers to the traceability of requirements from the requirements specification to segments of the system and visa versa.

Our work is more related to pre-RS traceability since we are not concerned with tracing code segments to the requirements document. Rather, we are concerned with explicit representation of domain knowledge that lead to the production of the requirements document. Nonetheless, it is important to note that our work is not targeted at pre-RS traceability of requirements. Rather we aim to provide a way to preserve domain knowledge used in requirements engineering. However, documenting domain knowledge obtained in pre-RS processes, which can be facilitated by this research, can potentially facilitate pre-RS requirement traceability.

2.4 Knowledge Management Systems

The main premise of this thesis is that information systems code can be used to facilitate accruing domain knowledge in an organization. In this section we provide a brief review of the term Knowledge Management Systems.

Knowledge is a complex concept and a number of factors determine the nature of knowledge creation, and management (Nonaka 1994). The general assumption in the IS literature is that there is a distinction between knowledge, information and data (Fahey and Prusak 1998). In light of this, a common view is that data is raw numbers and facts, information is processed data, and knowledge is authenticated information (Vance and Enyon 1998). Another prevailing view is that knowledge is information possessed in the mind of individuals (Alavi and Leidner 2001, Fahey and Prusak 1998). However, in the
context of KMS, the view presented by Davenport and Prusak (1997) is often adopted (Hahn and Subramani 2000), as it seems to specifically address the purpose of KMS: "Knowledge is a fluid mix of framed experience, values, contextual information and expert insight that provide a framework for evaluation and incorporating new experiences and information".

Knowledge management processes and activities refer to identifying and leveraging the collective knowledge in an organization to help the organization compete (Von Krogh 98, Dixon 2000). Knowledge Management Systems are systems supporting knowledge management. Knowledge management is a broad term, which refers to both tacit and explicit knowledge. In this context, explicit knowledge is defined as articulated, codified and communicated in symbolic form and/or natural language (Alavi and Leidner 2001). Tacit knowledge refers to knowledge that has personal quality, is "in the mind of individuals", and is hard to articulate or communicate (Hahn and Subramani 2000).

Two broad classes of KMS approaches relate to these two types of knowledge, and are often referred to in the IS literature, these are the personalization and codification approaches (Hansen et al. 2000). The personalization approach addresses tacit knowledge and attempts to link people to enable the transfer of knowledge. One way to enable communication is to provide pointers to location of expertise in the organization. This method is exemplified by knowledge directories, sometimes referred to as "yellow pages" (Alavi and Leidner 2001). The codification approach, on the other hand, addresses explicit knowledge. In this approach knowledge artefacts are stored and indexed in databases that enable easy retrieval. However, it is important to note that tacit and explicit knowledge are
not dichotomous states of knowledge, but are mutually dependent and reinforcing qualities of knowledge (Alavi and Leidner 2001).

Davenport and Prusak (1997) indicate that one possible aim of knowledge management projects is to make knowledge visible and show the role of knowledge in an organization. We suggest that since large quantities of explicit knowledge are embedded in software, this knowledge can be considered part of the knowledge potentially available to the organization. However, often system code is not designed to convey this knowledge explicitly. This issue is closely related to the objective of our thesis.
3. Domain Knowledge Constructs

In this section we establish an understanding of the term domain knowledge as it appears in current literature. We begin by framing the general concept of domain knowledge, and continue by articulating domain knowledge gathered in RE.

3.1 Domain Knowledge in Software Development

Domain knowledge is a widely used concept which transcends the scope of IT. As such, it has different meanings in different fields of research. Even within a single area of research we might encounter diverse definitions of this concept. Computer scientists often characterize domains as software, sets of applications, or common functions occurring in multiple systems. Starting from this common viewpoint, however, there is still enough room for disagreement, and various ways of understanding and analyzing domains and domain models (Simos 1996). Indeed, when related to IT in general, the term “domain knowledge” is very loosely defined. In a broad context, a domain is “a sphere of knowledge, influence, or interest” (Webster Dictionary). Iscoe et al. (1991) recognize the existence of different types of knowledge in software and the difficulty to define domain knowledge (page 340): “Writing an application program requires an understanding of both programming knowledge and application domain knowledge. Programming knowledge is relatively well understood. It is formal, modeled in various ways, explicit enough for teaching its objectives to novices, and general enough to apply across many
domains. Domain knowledge, although it clearly exists in the minds of domain experts, is not so well understood”.

In software engineering, the Object Oriented view considers domain knowledge to consist of the real-world actors, objects and operations as opposed to program and data that will be the subjects of object-oriented design (Simos 1996). The AI view locates domains in the real-world problem space as well. However, in AI, inference strategies are also associated with real-world processes. Therefore, in AI the way the system operates also deals with domain knowledge (Simos 1996). Nonetheless, in software engineering approaches domain knowledge is not possessed by programmers. The term “domain expert” is likely to connote a nonprogrammer practitioner (Simos 1996).

In line with the idea that domain knowledge in IS goes beyond the subject world of programmers, when narrowed to the field of IS, domain knowledge has been described as “an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area” (Jacobson et al. 1999). This is most often considered as the application area for which we develop software systems (Ruben 1990). This makes domain knowledge highly related to RE. Specifically, obtaining proper domain knowledge is recognized in software development as a critical factor for accomplishing the goal of RE, namely identifying complete, consistent, and accurate needs (Sutcliffe and Maiden 1998). In other words, during the requirements engineering (RE) process domain knowledge must be acquired. However, throughout the RE process, and as system development progresses, different types of knowledge domains interact and relate to the domain knowledge acquired. These include, for example, knowledge about programming and data. In this context, Ruben (1990) notes that every piece of knowledge in software
represents knowledge associated with some domain. These domains include the real-world knowledge the end user has, the architectural or implementation knowledge the software developer has, and possibly other levels of knowledge as well. Further, the different knowledge domains are not isolated from one another but rather interact with each other. In this context, we may consider domains as networks in some semi hierarchical structure. In such a hierarchical structure RE domain knowledge would be located at the top of the hierarchy, while other types of knowledge interact and make use of this knowledge. “Domains with narrow scope can sometimes be a part of broader domains. In such a structure primitive, narrow domains, such as assembly language and arithmetic operations, are at the bottom and broader. More complex domains are at the top.” (Ruben, 1990)

In short, while the exact scope of the term “domain knowledge” in software engineering is not clear, effectively collecting this knowledge is a critical aspect of RE. Moreover, other knowledge domains, pertaining to system development and implementation, are related to the domain knowledge acquired during RE. In the next section we analyze the latter.

### 3.2 Knowledge Types in Requirements Engineering

Domain knowledge is primarily accumulated during the early stages of the RE process. In the early phase of RE a considerable body of knowledge is built-up (Yu 1997). This stage of RE is intimately related to Systems Analysis – the process of analyzing the organizations and its needs. Korhonen (2000) notes the analysis phase is often described as defining the “what” of the software, whereas design defines the “how” of the system.
Related to that, Jacobson et al. (1999) state the “what” consists of an analysis of real-world business objects.

An important factor of domain knowledge highlighted by Zave (1989) is its formalization through a model. This task is accomplished by applying domain modeling, or conceptual modeling. An important class of conceptual models are Enterprise Models (EM). (Bubenko 1980) defines an enterprise model as “a representation of the organizational domain for which the information system is developed”. Enterprise modeling has emerged as a preliminary design phase in software systems development to capture the most important aspects in an organization (Thompson 1993, Castano et al. 1998).

While RE domain knowledge (hence on we use the abbreviation REDK) is captured in models during the early stages of RE this process is not clearly distinguished from other RE activities. A wide breadth of knowledge is included in the process of RE. This knowledge includes definitions of domain terminology, system organization, and non-functional characteristics. Or, as stated by Lamsweerde (2000, p6), “the scope of the requirements engineering process is fairly broad as it ranges from a world of human organizations or physical laws to a technical artefact that must be integrated in it.” This issue makes the identification of domain knowledge in the RE process a challenging task. However, by identifying the core domain-related constructs which are analyzed and dealt with during RE, the scope of REDK can be defined. In the next section we apply such an approach.

3.3 Developing the Model of REDK

In this section we develop a model of REDK to be explicitly represented in software. We begin by analyzing the contents of REDK as presented in the literature.
analyze whether there are missing constructs implicitly used in the literature, or if some
constructs have implicit characteristic which require further analysis. This process leads to
a final REDK meta model. Finally, we will turn to ontology to guide us in the process of
formalizing REDK constructs for representation in software.

3.3.1 Basic Model of REDK

In developing the basic model of REDK we take the approach of meta-modeling
constructs used in the different RE approaches. Rather than impose a meta model of the
domain related constructs we take a step by step approach to show the origin of our meta
model, and how it relates to different streams of research. To accomplish this, we analyze
foundational research in each RE modeling approach. From this research we extract the
core constructs described in each stream. We then construct, for each modeling approach, a
meta model of the core modeling constructs. Finally, we unify all the different meta models
to one overarching meta model. We examine each construct in each meta-model and
compare it to benchmark constructs used in Enterprise Ontology (Uschold et al. 1998). We
employ this approach to provide clear traceability to the modeling constructs covered, how
they relate to different literatures streams, and how they relate to one another when unified
in one overall modeling construct. We then check their fit in other RE literature. We find
that the domain constructs used in a sample of other RE cases matches our constructs.

We use a modified ER notation to represent our meta-models. Specifically, in our
meta-models, constructs are depicted using entity type symbols. Relationships between
constructs are depicted using relationship type symbols. The order of reading the
relationship is indicated by a small arrow placed adjacent to the diamond shape. A solid
arrow indicates an is-a relationship between elements, where the arrow is directed form the more specific entity towards the more general entity. Additionally, composition is notation is barrowed from UML. I.e. composition is indicated using a small composition diamond connecting between the related entities. Finally, a dashed arrow indicates a manifestation relation between two elements. I.e. when one element is used to represent another element, a dashed arrow is directed towards the former from the latter.

3.3.1.1 Domain Knowledge in Data-Based Approaches

Since information systems use and generate large volumes of information, the information needs to be understood, manipulated and managed. Data modeling aims to help stakeholders make decisions about what information the system should represent, and the correspondence between the information system and the real world (Nuseibeh and Easterbrook 2000).

One of the better known data-based approaches is ER modeling (Chen 1976). ER diagrams consist of Entities, Relationships, and in the extension termed E-R-A, also Attributes. The ER model has evolved over time, and today the Extended Entity Relationship model (EER) is often referred to in conceptual modeling. Basically, the EER model extends the ER model to incorporate generalization and specializations relationships.

Similar to ER, object-oriented techniques include objects, attributes and structural relationships – Generalization and Composition. A significant difference from the EER approach is that in the Object Oriented paradigm objects additionally incorporate services/methods. As well, Object Oriented methods support Polymorphism and Encapsulation. Polymorphism means that different classes may have different meanings/implementation for the same method. Encapsulation means that different classes
can interact without knowing the internal detail implementation of one another. That is, different classes interact only through a predefined interface.

Johnson et al. (1992) describe the main constructs used for domain modeling: types, instances, relationships, events, and invariants. In their work, types, instances and relations relate to the traditional E-R model, with few modifications freeing them from limitations associated with the fact that constructs are usually derived from programming languages. Events subsume all actions occurring in the system or the environments. Events have duration, possibly spanning multiple states in succession and may involve multiple entities. Events have preconditions, post-conditions and methods consisting of procedural steps Johnson et al. (1992). They may be explicitly activated by other events or automatically activated when their pre-conditions are met. Invariants are predicate conditions that are required to hold at all times or when an event takes place. Another modeling method, which concentrates on data modeling, is the Object-Role-Modeling (ORM) approach (Halpin 1998). ORM views the world in terms of objects that play roles. In essence, in ORM objects must have a name and play roles in relationships. That is, in ORM the way an entity participates in a relationship is its role. To this end, ORM defines relationship types, which are can be either relationships as defined in ER, or elements linking entities to other entities or to values. Relationships can be of various arties. That is, a relationship can be defined to include virtually as many roles as desired. However ORM does not include attributes explicitly in its reference model. Similar to the EER approach, ORM also defines cardinality constraints on relationships. That is, they provide a way to state that a relationship is restricted on the number of entities that participate in it. However, since attributes are not defined, cardinality constraints on attributes are, of course, not included.
The domain aspects captured in the data model approach are depicted in figure 1.

![Diagram](image)

**Figure 1: Meta model of data-based modeling constructs**

### 3.3.1.2 Domain Knowledge in Behavioral Approaches

As stated in section 2.1.1.2 Behavioral modeling approaches concentrate on the way the system behaves. As such, behavioral modeling approaches emphasize the system rather than the domain. Still, the domain in which the system operates is considered in behavioral modeling approaches since the interfaces between the system and the domain are analyzed.
in these approaches. Specifically, in the behavioral perspective, the attempt is to identify the interesting events that occur in the real world, the information affected by their occurrence and the functions that are invoked and cause this effect to take place. For this, three things are done (a) real events are abstracted into information bearing events, (b) real world things are abstracted into information bearing entities and relationships, and (c) functions to be invoked to carry out the effect of the event are identified and associated with it (Rolland and Prakash 2000).

In general, Rolland and Prakash (2000) suggest that there are three traditional categories of behavior models:

1. Control flow is the oldest and most popular form of behavior model. Control flow iterates through steps upon completion of a previous step, regardless of the availability of inputs. Only upon transitions between steps are inputs determined. As well, in control flow there are no restrictions on how inputs are given. For example, a postal delivery person starts rounds, regardless of whether mail needs delivery, because there may be mail to pick up.

2. Data flow modeling technique is the basis of functional languages. Data flow iterates through steps upon input provision. To this end, inputs are provided directly from outputs of other steps. For example, in factory manufacturing, each step is carried out when the developed product arrives from other assemblies.

3. State machines are the basis of UML's behavioral models. State machines iterate through states upon events in the environment. The inputs to a state transition are calculated as part of the step itself. For example, in a Point of Sale system, scanning a product displays product information on to the customer.
The three types of behavioral models emphasize different things: Control flow stresses sequencing of steps. That is, in control flow a step must be completed before a consecutive step is performed. Moreover, these models are not concerned with calculation of inputs using available information upon entering a state.

Different from that, Data flow requires the outputs of one step to be explicitly linked to inputs of consecutive steps. This stresses calculation of inputs, but largely puts aside the sequencing of steps, since whenever inputs are ready steps are traversed. Hence, data flow are similar to control flow but are more suitable when proper determination of inputs is most important, for example in manufacturing systems.

Finally, the emphasize in state machines is the response to an event. In state machines states are traversed when events happen in the environment of the system. Thus, both sequencing and calculation of inputs to each step are not emphasized in state machines. Calculation of inputs can be part of the steps themselves, and sequencing can basically be modeled as an event. Hence state machines are different from control flow and data flow, as they place emphasize on the environment. With state machines a system operates in response to its environment. Notably, in the behavioral approaches states are of the system and not the environment. However, since the system incorporates a representation of the environment, some of these system states may map to states of the environment.

While the first two categories are more related to the inner working of the system, the state machine category also relates to the environment since system states can reflect the environment states, and since events originate from the environment.
Underlying most of the state-oriented techniques are finite state machines. A finite state machine is a hypothetical machine that can be in only one of a given number of states at any specific time. In response to an event the machine operates and generates state changes.

Different diagrams are used to describe state machines. Two notations commonly used are state transition diagrams and state transition matrices. Basically the former models state machines using graphs with nodes representing states and arcs representing transitions. The latter models state machines using tables, in which rows denote possible origin state, columns denote stimuli, and cell content denote the next state. When considering the system outputs originating from events either a Moore or a Mealy model of the diagram can be considered (Kain 1972). In a Moore model system responses are associated with the state rather than the transition between states. In contrast, in a Mealy model the system response is a function of the transition. It should be noted however, that the only output of interest when considering the domain is the output which may affect state changes in the system entities which reflect the environment. That is, any change to the environment should be reflected in the representation of the environment embedded in the information system. For example, if as an output the system manufactures an item, a state change should result in the system to reflect this notion. However, if as an output the system prints on the screen, no state change within the entities representing the environment would occur.

Within the context of domain knowledge elements, it is sufficient to consider state changes and events in the environment. Under such conditions, there is no difference between Moore and Meely machines. Moreover, Moore and Mealy machines are identical
with respect to their power as trivial algorithms exist to transform either one into the other (Kain 1972).

The use of state machines has been transformed and applied in UML to accommodate object-oriented notions. UML attempts to unify all three traditional kinds of behavior models: control flow, data/object flow, and state machines (Bock 1999). Since UML is designed to be applied throughout the development process such unification is appealing. However, in order to examine domain knowledge in behavioral models, the underlying basic state machine models should be analyzed. State machines serve as the underlying models of many requirements engineering methods, including Statecharts, and Petri nets.

Statecharts (Harel 1987) extend state machines by providing notation and setting conventions that facilitate the hierarchical decomposition of finite state machines. In statecharts, a transition can be a function of not only an external stimulus but also of the truth of a particular condition. For example, a transition from dial tone to busy tone states would occur as a response to an event of dialing a number AND the condition that the party being called is active. I.e. not only the stimuli determines state changes but also some condition. As well, Statecharts introduce Superstates. Superstates can be used to aggregate sets of states with common transitions. For example, if we assume that two states S1 and S2 both transition to state S3 on the same stimulus, Superstate S4 may be defined as S1 OR(AND) S3. Finally, Statecharts provide default entry state which is the one of the states a superstate into which the state machine enters when a transition bringing the system to the superstate takes place.

Petri nets were first introduced by Petri (1962) and are well described by Peterson (1977). Petri nets are virtual machines with a well-defined behavior. They have been
initially used to specify process behavior during the design phase of time critical applications. Later they have been applied during requirements analysis as well. Petri nets model a system in terms of transitions and places, which can have tokens. Transitions can consume tokens in some transitions, and generate tokens in others. The core constructs used in behavioral modeling are depicted in figure 2.

![Diagram of behavioral modeling constructs]

**Figure 2: Meta model of behavioral modeling constructs**

### 3.3.1.3 Domain Knowledge in Goal Oriented Business Enterprise Modeling Approaches

Osterwalder et al. (2005) conducted a literature review regarding the use of the term “business model”. The work revealed that there is a much variance between authors, some using the term to simply refer to the way a company does business and others emphasizing
the model aspect. The former generically refer to the actual business activities in a company; whereas the latter refers to a conceptualization of the way a company does business in order to reduce complexity to an understandable level. Proponents of the latter viewpoint propose meta-models that consist of elements and relationships that reflect the complex entities that they aim to describe. In other words, under this view of business models, the quest is to identify the elements and relationships that describe the business a company does.

With respect to enterprise modeling, the term is a collective name for the use of models in enterprise engineering and enterprise operation (Bernus 2001). Therefore, in our context, Enterprise Model and Business Domain Models refer to the same construct.

Wand et al. (1999) have defined the enterprise model of a business to distinguish it from other concepts/models such as business architecture, IS architecture, and IS implementation. They state that a business domain model reflects the core business of an organization. The business domain model is the starting point for designing information systems (Wand et al. 2000). This model is based on the notion that a company delivers products and services to customers. The business model should capture, at a very high level, how an organization behaves to satisfy the customers’ requests for products/services. The business model does not show how the organization is structured and operates in order to accomplish its goal. These aspects, which they term “business architecture, are considered an implementation of the business model.” (Wand et al. 1999 p. 5). Moreover, Wand et al. (1999, p.5) explicitly note that objects in the business model are not processes, specific agents, or database components. Rather, they are roles played by agents (e.g.,
people and departments) responsible for delivering services/products among themselves and, eventually, to customers.

In Wand et al. (1999) a methodology for constructing a business model in the process of systems analysis/requirements engineering is described. This methodology can help understand the type of business domain knowledge gathered during the early steps of the RE process. The methodology described by Wand et al. (1999) is defined as request-driven. The business model is developed by decomposing it into roles based on the requests with which the business has to deal. When employing the methodology, external objects, which send requests to the organization, are identified. In other words, objects that interact with, but are independent of the organizational system and play no roles in carrying out services within the organization. Examples of external objects can be customers, students, suppliers, etc. The methodology further associates services in the organization to process requests. Objects are assigned for delivering services. The methodology further identifies object interactions. An interaction occurs when an object needs a service of another object in order to provide its own service. In the case of interaction, internal requests are sent from one object to another.

In the 90s work on Enterprise modeling evolved to become goal-oriented. In this context, Yu’s model (1997) emphasizes the difference between early phase requirements and late phase requirements. In the early phase of requirements engineering, the different activities address organizational or non-functional concerns. That is, the motivation and rationale that underlie system requirements is addressed. In the late phase of requirements engineering, issues such as completeness and consistency are addressed. The i* modeling technique (Yu 1997) provides an understanding of the enterprise by focusing on “why”
questions for the information systems considered. In i* the participants in the organizational setting are actors with goals. These actors depend on each other in order to fulfill their objectives and have their tasks performed. The i* technique offers two types of models: The Strategic Dependency (SD) model, and the Strategic Rationale (SR) model (Yu 1997).

The SD model focuses on the intentional relationships among organizational actors. The SR model describes stakeholders’ interests and concerns and how they might be addressed by the various configurations of systems and environments. The Strategic Dependency model describes the dependency between various actors in the organization. It consists of a set of nodes and links connecting them, where nodes represent actors and each link indicates a dependency between two actors. The depending actor is called depender, and the actor who is depended upon is called the dependee. i* distinguishes three types of dependencies: goal dependency, resource dependency, task dependency. In this context, a goal can be a softgoal as well – i.e. a qualitative goal, for which it is hard to determine if it has been satisfied. In a goal dependency, an agent depends on another to fulfill a goal, without worrying how this goal will be achieved. In a resource dependency, an actor depends on another actor to provide a resource. In a task dependency, an agent depends on another to carry out a task. In i* the SD model examines the organizational needs and dependencies.

The SR models provide a complementary view of actors by looking “inside” actors to model internal intentional relationships. In particular, such models describe the alternative ways goals can be satisfied or tasks can be accomplished, and offer rationale for various actor dependencies. SR models include goal, task, resource and soft-goal. In addition, they
use means-end relationships which describe how to decompose a goal into subgoals. As well, they offer task-decomposition relationships to describe task decompositions into subtasks. For example, a task of giving an exam at a university setting may be decomposed to write exam, scheduling exam, and handing out exams.

It should be noted that goal-oriented requirements engineering methods such as the i* modeling framework, are closely related to agent-oriented models. For example, the ALBERT language (Dubois et al., 1993) is an agent-based requirements language, which has been designed for specifying the requirements of distributed real-time systems. Work has been conducted on the combination of i* with ALBERT (Yu et al. 1995). As well, i* is the basis for Tropos, a requirements-driven agent-oriented development methodology (Castro et al. 2002). The Tropos methodology guides the development of agent based systems from the early requirements analysis.

Another popular goal-oriented approach is KAOS (Dardenne and van Lamsweerde 1993). In KAOS, requirements specification consists of progressively refining high-level goals until constraints, objects and operations that are assignable to individual agents are defined. The KAOS approach consists of a formal framework based on temporal logic and AI refinement techniques where all terms such as goal and state are consistently and rigorously defined. The main emphasis of KAOS is on the formal proof that the requirements match the goals that were defined for the envisioned system.

A KAOS specification consists of three core models: *goal model* where goals are represented, and assigned to agents; and *object model*, which can be derived from formal specifications of goals; and an *operation model*, which defines various services to be provided by software agents.
Within the goal model, goals are defined and analyzed. A goal in KAOS is defined in Lamsweerde (2004) as a "prescriptive statement of intent about some system whose satisfaction in general requires the cooperation of some of the agents forming that system". Goal definition patterns are used for specification of goals at the modeling layer. These are specified in temporal logic and include patterns such as achieve, cease, maintain, optimize, and avoid. KAOS also supports additional types of goals (Dardenne and Lamsweerde 1993). For example, satisfaction goals are functional goals concerned with satisfying agent requests; information goals are also functional and are concerned with keeping such agents informed about object states; accuracy goals are non-functional goals that require that the states of software objects accurately reflect the state of the observed/controlled objects in the environment (Dardenne and Lamsweerde 1993). Non-functional goals usually deal with the system rather than its environment. More categorization of such goals is discussed by Chung et al. (2000).

Goal categories form a hierarchy, in which at the root are system goals and private goals. System goals, however, only correspond to the system and do not incorporate domain goals.

All goals are organized in KAOS in the usual AND/OR refinement hierarchies. Goal refinement ends when every subgoal is realizable by some individual agent assigned to it. To be realizable the goal must be expressible in terms of conditions that can be monitored and controlled by the individual KAOS agent.

In the object model, KAOS includes objects, which are very similar to those discussed in data model approaches. Objects can be entities, relationships, or events. KAOS also includes Operations, described in the operations model. Operations are input-output
relations over objects, and describe behavioral aspects. In essence, operations define state transitions. Operations are declared by signatures over objects and have pre-, post-, and trigger conditions. KAOS makes a distinction between domain pre-/post-conditions for an operation and desired pre-/post-conditions for it. Domain conditions describe what an application of the operation means in the domain, without any prescription as to when the operation must or must not be applied. Desired conditions capture additional conditions to ensure that the goals are met (Letier and Lamsweerde 2002).

In KAOS, agents are also defined. An agent is a kind of object that acts as a processor for operations. Agents are active and can be humans, devices, software, etc. KAOS enables the specification of which objects are observable or controllable by agents.

Finally, the GBRAM (Anton 1996) method also uses goals as the means to elaborate and structure system requirements. Similar to the other goal-oriented approaches, a system and its environment are represented in GBRAM as a collection of agents, which are defined as entities or processes that seek to achieve goals within an organization or system based on the assumed responsibility for the goals.

GBRAM offers prescriptive guidelines on how to extract goals from different sources into one ordered goal set. GBRAM defines a top-down analysis method refining goals and attributing them to agents starting from inputs such as corporate mission statements, policy statements, interview transcripts etc. The operationalized goals, responsible agents, stakeholders, scenarios and obstacles provide a representation of system requirements organized according to system goals. To accomplish these objectives GBRAM involves goal analysis and goal refinement.
Goal analysis is about the exploration of information sources to identify goals, followed by organization and classification of the identified goals. Goal refinement concerns the evolution of goals from the moment they are first identified to the moment they are translated into operational requirements used in the system specifications.

This activity is followed by refinement activities that involve the pruning of the goal set (e.g., by eliminating "synonymous goals"), by analyzing the goal set and considering possible goal obstacles (Potts 1995) - other goals that prevent or block the achievement of a given goal. As well refinement in GBRAM includes the identification of goal precedence. This can be achieved by searching for agent dependencies. For example, in Anton (1996), if a supervisor depends on the employee to provide him with a time sheet in order to approve a weekly payment, then there is an agent dependency. This dependency reflects the goal precedence as well.

Finally, in GBRAM a requirement specifies how a goal should be accomplished by a proposed system. GBRAM also defines Constraints. Constraints provide additional information regarding requirements that must be met in order for a given goal to be achieved.

A meta model reflecting the additional domain constructs supported in goal-oriented modeling is depicted in figure 3. Note that in this figure, we have not included elements overlapping with the behavioral and data-based models, covered in the previous section. Also note that in the figure there are some verbs defining a few domain entities. Specifically, following KAOS, different goal types are defined using verbs.
3.3.1.4 The Domain Knowledge Approach in Requirements Engineering

An approach appealing to our research views RE mainly as a knowledge modeling process. This approach is taken, for example, in Johnson et al. (1992). This approach was also taken by Kavakli and Loucopoulos (1999), who suggested that modeling of organizational change encompasses the following four concerns:

1. Understanding the current enterprise situation.
2. Knowledge about how change can take place.
3. Knowledge about the future enterprise system.
4. The concern of evaluating enterprise models against the criteria of the parties involved.

These four types of knowledge bring the RE process to four different states of knowledge respectively (Kavakli and Loucopoulos 1999):

1. The As-Is knowledge state,
2. The Change knowledge state,
3. The To-Be knowledge state, and
4. The Evaluation knowledge state.

The two knowledge states, in which knowledge about the environment is established, are knowledge states 1 and 3. Domain knowledge consists of As-Is knowledge, which remains valid after the change process. It also consists of To-Be knowledge, which relates to entities that are part of the business environment.

It is interesting to note that the order of transition between the states is not strictly defined (Kavakli 2002), but rather the employed RE methodology defines the order in which the different knowledge states are traversed. I.e. the order of arrival at different knowledge states, As-Is knowledge state, Change knowledge state, To-Be knowledge state, and Evaluation knowledge states, is different in different RE methodologies.

While goal-oriented methods put more emphasize on the domain than behavioral and data modeling approaches, domain modeling approaches take the stand that even more emphasize should be placed on the domain rather than on the system. According to this approach goals by themselves do not make a good starting point for requirements engineering. Zave and Jackson (1997) illustrate this by considering a project to develop a computer-controlled turnstile guarding the entrance to a zoo. They show that domain
knowledge is required to define the scope of relevant goals. If the engineers are told that the goal of the system is to deny entrance to people who have not paid the admission charge, it may be suggested that the goal has been stated too narrowly as perhaps the real goal should be to ensure the profitability of the zoo. Thus perhaps the engineers should consider other ways of improving profits, such as cutting costs. Following this line, it may be good to consider whether more money can be made by closing the zoo and selling the land; and so on (Zave and Jackson 1997). Basically, almost every goal is a subgoal with some higher purpose. Zave and Jackson therefore highlight the need to have a clear relationship between requirements and specifications. This relationship talks about the mediating effect of domain knowledge between specifications and requirements.

From the above it follows that specifications can be said to satisfy requirements only when incorporating domain knowledge. The domain knowledge basically guides us to the scope of relevant domain. More specifically, a specification together with relevant domain knowledge should be sufficient to guarantee that requirements are satisfied. This is formalized in Zave and Jackson (1997):

\[ K, S \models R \]

Where \( K \) is a description of the problem domain, \( S \) is the specification of the solution, and \( R \) is the problem requirement. Parnas and Madey (1995) take the approach that requirements are in essence constraints imposed on the environment. They define the relation they designated as REQ, which incorporates constraints on the environmental quantities. Specifically Parnas and Meday define in the System Requirements Document environmental quantities, which are measured or controlled by the computer system. This document includes a specification on each environmental quantity as either monitored (a
quantity that the system needs to measure), controlled (a quantity that the system needs to control), or both. The environment knowledge incorporated in this document can be described by a relation defined by Parnas and Madey as the NAT relation (standing for Nature):

- Domain(NAT) is a set of vectors of time functions containing monitored values at different times
- Range(NAT) is a set of vectors of time-functions containing the values allowed by controlled variables.

Work viewing RE under the domain knowledge approach dates back to 1982 with the work of Dubois et al. (1986), who suggested the Entity-Relationship-Attribute-Event, (ERAE) language. This language borrows from ideas in Semantic Networks and Logic. The Language uses a semantic network type of graphical notation, and supports the use of the fundamental Entity Relationship constructs as well as the construct of Event. To this end, ERAE is also proposed as a possible language to help derive requirements for further validation, basing them on available requirements and temporal logic.

Today this view is mostly reflected in the Problem Frames method (e.g. Hall et al. 2005). As the name implies, Problem Frames structure the problem domain and describe the effects of the system on the problem domain. By emphasizing problems rather than solutions, Problem Frames can exploit the understanding of a problem class, allowing a problem owner with specific domain knowledge to drive the RE process (Hall et al. 2005). The unique approach of problem frames is in the clear attempt to distinguish between system and environment requirements. The roles of descriptions in the framework are twofold. 1) indicative descriptions express what is assumed to be true in the problem
frame. and 2) *operative descriptions* express what is desired to be true once using the system. According to this, given domains have an indicative role, while the requirement description and machine specification have an optative role. In the context of our work, indicative descriptions about the domain are naturally the descriptions which relate to domain knowledge.

In the Problem Frame framework, a *problem diagram* captures the characteristics of the problem domain as well as requirements that basically constrain the domain.

Within a problem frame a *machine domain* is defined. The machine domain is the system to be built together with its underlying hardware. In contrast to the machine domain, other domains, termed *given domains*, represent parts of the world that are relevant to the problem. These domains include physical events and states that are causally related. The different domains may share events and state information. These are called shared phenomena in the problem frames framework. Phenomena shared between two domains are observable by both, but controlled by one of them only.

Operators in the problem frame framework are the human operators. They are termed as *biddable domains*, meaning that they may obey stipulated procedures, but not reliably, and they may generate events spontaneously.

In the problem frames framework, a requirement is defined as a condition in the problem domain that the machine domain must guarantee to qualify as a solution to the problem. The phenomena of biddable domains are events; those of causal domains are states and events (generalized as causal phenomena).

The domain constructs of the domain modeling approaches are summarized in the meta model depicted in figure 4.
3.3.1.5 Integration

In this section, our purpose is to create a meta-model of RE domain knowledge that will encompass the four RE approaches, namely the data based, the behavioral, goal-oriented, and domain based. To combine the domain knowledge elements included in the four models, we suggest creating a mapping of these constructs to those of the Enterprise Ontology proposed by Uschold et al. (1998). Uschold et al. (1998) provide a generalized ontology of domain concepts used in business. We are interested in requirements engineering domain concepts, which are only part of the ontology of business concepts.
defined at Uschold et al. (1998). In this way, the enterprise ontology serves as a benchmark for determining whether two knowledge elements in different meta-models relate to the same domain concept. For example, we find that the terms event and transition relate to sub elements of the same construct in the Enterprise Ontology, the transition construct. We also find that roles relate to the relationship construct, as roles are constructs manifesting interaction in goal based RE and the relationship constructs enables interaction between entities in data-based RE. Similarly, dependency relates to the same constructs as it too manifests interaction between elements. The full mapping is shown in table 1.
Table 1. Requirements Engineering Domain Knowledge Constructs Mapping.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Goal Oriented</th>
<th>Data based</th>
<th>Behavioral</th>
<th>Domain Based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actor</strong></td>
<td>Actor (e.g. Anton 1996)</td>
<td>Entity (e.g. Rumbaugh et al. 1991)</td>
<td>-</td>
<td>Operator Domain (e.g. Hall et al. 2005)</td>
</tr>
<tr>
<td></td>
<td>Agent (e.g. Castro et al. 2002)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Role (Actor Role)</strong></td>
<td>Role (e.g. Wand et al. 1999)</td>
<td>Role (e.g. Halpin 1998)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dependency (e.g. Yu 1997)</td>
<td>Relationship (e.g. Chen 1976)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Goal / Subgoals</strong></td>
<td>e.g. Achieve/Goal/Maintain (e.g. Dardenne and Lamsweerde 1993, Castro et al.</td>
<td></td>
<td>-</td>
<td>Goal (e.g. Kavakli and Loucopoulos 1999)</td>
</tr>
<tr>
<td><strong>Purpose / Help Achieve</strong></td>
<td>(e.g. Dardenne and Lamsweerde 1993, Castro et al. 2002)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Service (Activity)</strong></td>
<td>High level Task (e.g. Yu 1997)</td>
<td>Operations/Methods (e.g. Coad and Yourdon 1990)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Resource</strong></td>
<td>Resource (e.g. Yu 1997)</td>
<td>Entity (e.g. Chen 1976)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Constraint (Activity Specification)</strong></td>
<td>Cease Goal; Avoid Goal (e.g. Dardenne and Lamsweerde 1993)</td>
<td>Cardinality/Invariants</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>State (State of Affairs)</strong></td>
<td>-</td>
<td>State (e.g. Petri 1962)</td>
<td></td>
<td>State (e.g. Zave and Mylopooulos 1999)</td>
</tr>
<tr>
<td><strong>Transition (incorporate events)</strong></td>
<td>Request (e.g. Wand et al. 1999)</td>
<td>Precondition (e.g. Johnson 1993)</td>
<td>Transition (e.g. Petterson 1977) Event/Condition (e.g. Harel 1987)</td>
<td>Causal Phenomena (e.g. Hall et al. 2005)</td>
</tr>
</tbody>
</table>
As shown in table 1, based on the RE literature and the Enterprise Ontology, we identified nine fundamental domain related constructs: Actors, Roles, Resources, Services, Goals, Constraints, Transitions, Events, and States (Consisting of Attributes). We now provide a short definition of these constructs, by referring to Uschold et al. (1998) ontology as a benchmark for definition.

**Roles:** Uschold et al. (1998, p.42-43): An entity may have a role reflecting its relationship with other entities. Some relationships are special in that they entail some notion of doing or cognition (e.g. performing an Activity, or holding an Assumption). We refer to an entity involved in such relationships as an actor. Relationships (interactions) among actors may entail some view (activity or cognition) for one of them. This view indicates the actor has an actor role (Uschold et al. 1998).

Note that the enterprise ontology distinguishes between two types of roles. One indicating a general relationship and one is Actor role – indicating a sense of activity or cognition. The different approaches in ER generally do not make this distinction, and we explore this issue further in the next subsection.

**Actors:** Uschold et al. (1998, p.45): Certain roles in a relationship are special in that the playing of these roles entails doing or cognition. These are called actor roles. Entities playing such roles are called actors. Hence, actors are active entities in the organization. An actor can either be a person, an organizational unit, or a machine which performs some activity (Uschold et al. 1998).
**Resources:** Uschold et al. (1998, p.50): The Role of an Entity in a relationship with an activity whereby the entity is or can be used or consumed during the performance of the activity.

**Services:** Uschold et al. (1998, p.47 define activities, which we map to services): something done over a particular time interval. An activity can have pre-conditions and effects, can be performed by one or more doers, can be decomposed into more detailed sub activities, may entail the use or consumption of resources, can have authority requirements, and may have an owner. Services (activities) are closely linked to a doer, which is an actor executing it, and possibly consuming or using resources for the activity (Uschold et al. 1998).

**State:** Uschold et al. (1998, p.64): a state is some kind of situation that can be thought of as holding or being true (Uschold et al. 1998).

**Goals:** Uschold et al. (1998, p.55): a goal is the role of a state whereby the Actor wants, intends, or is responsible for the full or partial Achievement of the State. A goal (achieve) is the realization of a state (state of affairs).

**Constraints:** Uschold et al. do not explicitly define the term constraints. However, they mention three types of constraints: 1) Effect - A State that is brought about by an activity. 2) Pre-Condition- a state required to be true in order for the activity to be performed. 3) Restrictions on the range of activities in the universe (Uschold et al. 1998).
Transition: Event and transition collapse into one construct. The constructs which relate to these in Uschold et al. (1998) are Precondition and Effect. Uschold et al. (1998, p.48): Precondition is a State Of Affairs required to be true in order for the activity to be performed (Uschold et al. 1998 p.48). Effect is A State that is brought about by an activity. To this end, precondition well reflects the notion of event from the RE literature. An event of the RE literature does not map to the Enterprise ontology since in the Enterprise Ontology an Event is an activity outside the scope of interest (Uschold et al. 1998, p.49): Further, various formalisms for modeling activities distinguish between event and other types of activities. An event is seen as outside the scope of interest of the model apart from its effects. In particular, the model will not recognize the doer, the duration, choice or control over its occurrence.

Following this analysis and integration, a meta model, incorporating the domain related constructs and their relationships as they appear in the literature, is presented in figure 5. Note that this model contains one more relationship above those described in the enterprise ontology. Namely, we link attributes and states. This is done to bridge the data view (which has attributes) and the other models, which use states.
Figure 5: Meta model of requirements engineering domain knowledge constructs

We note of the additional notation used in this figure – a composition is indicated using the UML notation of a small diamond, (e.g. note the small diamond connecting between a state and attribute – that is, a state is composed of attributes);

The meta model of figure 5 is based on sources that represent the most common approaches of RE literature. We further test it against a set of additional literature sources, examining if the constructs we defined map well in those sources as well. Table 2 provides a summary of this analysis. The first column lists the benchmark constructs from table 1. Other columns reflect the terms used for the same constructs in the compared RE related
literature. We include in each column work representing a different perspective about RE. The two most widely discussed approaches are the Data based approach and the Goal Oriented approach. Greenspan et al. (1994) relate to the Data based approach and discuss general concepts for formal requirements modeling; Johnson et al. (1992) analyze the Data based approach as a domain knowledge documentation activity; Bresciani et al. (2004) relate to the goal-oriented approach and discuss agent-oriented RE methodologies; We also refer to Kirikova et al. (1994) who suggest submodels of enterprise models to reflect full RE knowledge. In their work, the unification of the different submodels forms a concept submodel that serves as an “ontology” of the enterprise. We also refer to the work conducted by Fox et al. (1993), who analyze models as a source for answering commonsense questions about the enterprise.

In Table 2, the contents of cells indicate the alternative terminologies used.
Table 2. Requirements Engineering Constructs in a Sample of Additional Resources.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actor</strong></td>
<td>Actor</td>
<td>Instance</td>
<td>Entity</td>
<td>Concept-Individual/ Organizational Unit</td>
<td></td>
</tr>
<tr>
<td><strong>Role</strong></td>
<td></td>
<td>Type</td>
<td></td>
<td>Goal</td>
<td></td>
</tr>
<tr>
<td><strong>Goal</strong> / Subgoals (Purpose / Help Achieve)</td>
<td>Goal/ Subgoals</td>
<td>-</td>
<td>-</td>
<td>Goal</td>
<td>Terminal state /Non-terminal states</td>
</tr>
<tr>
<td><strong>Service</strong> (Activity)</td>
<td>-</td>
<td>Methods</td>
<td>-</td>
<td>Process</td>
<td>Activity</td>
</tr>
<tr>
<td><strong>Resource</strong></td>
<td>Resource</td>
<td>Instance</td>
<td>Entity</td>
<td>Concept-Non-human Resource / Material</td>
<td>Resource</td>
</tr>
<tr>
<td><strong>Constraints</strong></td>
<td>-</td>
<td>Invariants</td>
<td>Assertions</td>
<td>Rule</td>
<td>Caused state</td>
</tr>
<tr>
<td><strong>State (State of Affairs)</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Problem/ Opportunity /</td>
<td></td>
</tr>
<tr>
<td><strong>Transition</strong></td>
<td>Request</td>
<td>Precondition</td>
<td>Precondition</td>
<td>Causal Phenomena/</td>
<td>Activity-State Cluster/ Enabling State</td>
</tr>
</tbody>
</table>

Table 2 summarizes how the meta model constructs (figure 5) are reflected in the additional literature set.

We now briefly show some example of how the constructs have been used in compared literature:

An Actor is a physical, social or software agent (Bresciani et al. 2004). Actors/Agents are central to requirements engineering, as modeled agents incorporate domain knowledge, which eventually propagate to the information system (Lamsweerde 2000). Actors expose
services representing their responsibilities and capabilities (Wand et al. 1999). Actors are performers of different activities (Caetano et al. 2005) and have capabilities and goals. An actor is someone (a human actor) or something (an automated actor, such as an information system or a production machine) that can perform the actions required by an Activity. Actors represent people, systems (mechanical or computer based) or a combination of both (Caetano et al. 2005).

The State of actors and resources is characterized by the values of their attributes (Caetano et al. 2005). Or as stated in Wand et al: “An object type has state variables reflecting its physical state and knowledge” (Wand et al. 1999). A state of an actor or resource reflects its situation and is represented by attributes (Caetno et al. 2005).

Resources represent things such as materials or information (Caetano et al. 2005). A Resource is the input or output of an Activity/Plan/Service. Plans/Activities/Services reflect what happens in the business (Caetano et al. 2005).

Goals are strategic interests of actors or a measurable state that the organization intends to achieve (Caetano et al. 2005). A Goal corresponds to an objective the system should achieve through cooperation of agents in the software-to-be and in the environment (Lamsweerde 2000). Goals are achieved/satisfied through the execution of Plans/Activities. Actors execute plans, and the execution of a plan can be a means for satisfying a goal (Bresciani et al. 2004). Actors depend on one another for goals to be achieved, task to be performed, and resources to be furnished (Giorgini, et al. 2005.) This fact is manifested
through dependency - an *actor* depends on another *actor* to attain some *goal*, execute some *plan* or deliver a *resource* (Bresciani et al. 2004). Goals may either be associated with the *domain* or with the *system-to-be* (Lamsweerde 2001). A Goal under the responsibility of an agent in the system-to-be is a requirement. A goal under the responsibility of an agent in the environment is an assumption (Lamsweerde and Willemet 2000). Goals are states which an *actor* has desire to arrive at. When goals contribute to the accomplishment of other *goals* they are considered as subgoals.

*Roles* organize a process' activities into sets of operations associated with a given *actor* (Caetano et al. 2005). *Roles* are the observable behaviors of *actors* defined in a specific collaboration context. Thus, a *role* represents the extrinsic features of an *actor* when it collaborates with other business objects. (Caetano et al. 2005). *Roles* are played collaboratively by enterprise *actors* to operationalize the enterprise *goals* (Kavakli and Loucopoulos 1999). Hence, a role is meaningful only when relationships exist between entities (Steimann 2000, Chu and Zhang 1997).

A role is an abstract characterization of the behavior of a social actor (Bresciani et al. 2004). In addition, Albano et al. (1993) and Steimann (2000), claim that a role comes with its own properties and behavior. These imply a set of services and attributes an entity provides (Albano et al. 1993, Steimann 2000).

While the analysis has enabled the identification of a scope for REDK, some of the identified constructs still cause some confusion. Specifically, there appears to be some
inconsistency with the way ternary relations are approached in RE. Consider the following relation, depicted in figure 6:

![Diagram](image)

**Figure 6:** An Actor interacts with another actor to perform a service

This relation denotes that actors interact in order for services to be performed. We propose to resolve this by showing that a role is a manifestation. This is expressed as denoted in the meta model by the manifestation relation (see figure 7). This manifestation reduces the conceptual complexity of this relation. By using the role construct one can express complex relationships in the domain.

Note, however, that the Role manifestation depicted in this figure is practically achieved by using two roles – one role for each actor involved in the interaction.

![Diagram](image)

**Figure 7:** Interaction is manifested by the role construct
However, the same solution cannot apply for resource. A similar relation found at the REDK meta-model does not have such a manifestation. I.e., the idea that a resource is used or consumed by an actor in a service cannot be manifested by a single construct (see figure 8). The RE literature does not provide a construct similar to role to capture the interaction between resources and actors.

![Diagram](image)

**Figure 8: A Resource is used by an actor in a service**

Issues such as the above suggest that perhaps domain knowledge accumulated during requirement engineering can be addressed in a conceptually clearer manner. We consider this in the next section, where we suggest enhancements to the initial model (figure 5).

### 3.3.2 Enhanced Model of REDK

It is important to remember that the definition of REDK presented in the previous section is based on work associated with requirements engineering. The field of requirements engineering is primarily concerned with requirements elicitation rather than documentation of domain knowledge. In this section we suggest enhancements to the original model (figure 5), intended to support the use of the constructs for representing domain knowledge.
In the RE literature roles are primarily referred to as ways to represent interactions between actors. However, in the general modeling literature the concept of roles is not limited to actors; rather roles can be played by any entity (Steimann 2000). While in RE some elements of roles, such as services provided, are not applicable to resources, other aspects of roles are equally applicable to actors and resources. Aspects of roles in RE which apply to both are:

1. Their ability to represent different ways an entity is viewed by other actors in the organization (Uschold et al. 1998): Both actors and resources can be viewed in different ways by other actors of the organization. For example, a specific room in a building is a resource which can be viewed as an office, or as a potential meeting area. Similarly, a specific person in an organization is an actor which can be viewed as a manager or as employee.

2. Their ability to engage in accidental relationships with some other entities (Sowa 1984, Guarino 1992): Actors can have temporary relationships both with other actors and with resources. Similarly, Resources can have temporary relationships with actors that use them. For example, an actor may have a temporal relationship with some computer assigned to him/her. Similarly, an actor can have a temporal relationship with some other actor assigned to work with him/her on a specific project. A resource, such as a stool, may have a temporary relationship with an actor using it. It may be used by one actor as a sitting item. By another as a heightening item.
3. The ability of an entity to play different roles simultaneously (Richardson and Schwartz 1991, Renouf et al. 1995, Steimann 2000): Both an actor and a resource can play different roles simultaneously. For example, a specific person can be both a student and an employee. Similarly, clothing items may be viewed as fashion accessories and warming facilitators simultaneously.

4. The fact that a role encapsulates a set of attributes (Albano et al. 1993, Steimann 2000): Both actors and resources have attributes that can be grouped under specific views. For example, a specific actor’s salary, average income, and occupational skills have to do with the actor’s role of an employee. Similarly a specific resource’s floor area and sitting capacity have to be with the resource’s role as a meeting room.

On the other hand, aspects of roles which apply only to actors but not to resources are:

1. Their ability to encapsulate behavior (Brensciani et al. 2004, Albano et al. 1993, Steimann 2000): The general convention in RE literature is that resources do not provide services or expose behavior. Entities which have such characteristics are considered actors.

2. Their ability to provide means of interaction with actors (Uschold et al. 1998): While actors interact with one another, actors are not considered as interacting with resources. Rather actors use or consume resources (Fox et al. 1993). While a relationship between resources and actors exists, it is not of interaction.

The correspondences between resources and roles on the one hand, and actors and roles on the other hand suggest that actors can play roles which have additional capabilities than
roles that resource can play. In other words, it can be said that two types of roles exist in the domain – roles that can be played by actors, and roles that can be played by resources. Further, since the properties of actor roles are a superset of resource roles, a generalization relation seems to exist between actor roles and resource roles, where the former are more specialized. By augmenting the REDK model to include resource roles we strengthen REDK model in two important ways. We enable resources to have different roles, and we provide means for explicitly representing the relationship between actors and resources in services.

The incorporation of these ideas into the REDK meta model results in the enhanced meta model depicted in figure 9:

![Diagram of Enhanced REDK Model](image)

Figure 9: Enhanced model of requirements engineering domain knowledge
4. Approach for Representing Requirements Engineering

Domain Knowledge

In this section we provide a way for representing REDK domain knowledge constructs. Our approach is ontology based and object-oriented. The use of object constructs can facilitate the representation of REDK in Information Systems code. The approach and the outcome representation objects are described in this section.

4.1 Approach Explained

We are looking for a way to represent REDK explicitly in IS code. By representing REDK in system code, the ability to understand the domain using system code may be improved. One could examine knowledge associated with the domain using constructs commonly applied in Requirement Engineering methodologies. Another benefit stems from enabling the validation that an information system is a good representation of the application domain, as defined by Wand and Weber (1988). More specifically, Wand and Weber suggest a set of requirements for an information system, necessary and sufficient for an information system to be a good representation of reality. These requirements all relate to a mapping between the real world defined as a triplet \( <S,L,E> \) (S is the possible state space, L is a set of system laws, and E is a set of external events) and an information system defined as \( <M,P,T> \) (M is the set of possible states, P is the information system law, and T is a set of external events). Having state variables, events, and laws explicitly represented enables validation that the requirements set by Wand and Weber are met.
In our approach we use the widely applied OO paradigm to enable the representation of REDK constructs. Further, Objects are introduced with ontological representation guidelines in order to have a coherent and well defined representation.

Generally, we suggest that using the REDK meta model constructs, explicit representation of REDK can be enabled. Taken to IS code, we will suggest a representation of REDK constructs using objects to enable direct use of the meta model constructs in IS code. This in turn should enable system developers to directly document in code the specific actors, resources, roles, goals, services, and constraints in the organization's domain.

For example, when using our representation in IS code, in correspondence with the meta model constructs depicted in figure 9, the construct Actor would be supported directly in system code. This construct will include Active Roles and Goals, which can be further referred to explicitly in IS code. Additionally, in correspondence with the meta model, the Active Role construct would include other meta model constructs such as Intrinsic Attributes, Mutual Attributes, and Services. Similarly, Resources, Roles, Constraints, and all other meta model constructs can be supported. These constructs can be employed when representing REDK in system code.

In general, regardless of where our representation constructs are used, whether in IS code or in other documents, using these constructs one would be able to explicitly represent the knowledge about a specific organizational domain. For example, one would be able to define a specific role in the organization's domain, such as a student; its attributes, such as GPA and Registered Courses; and its Transition Laws, such as transitions between states taking place while registration (Registered Courses become non-empty).
4.2 Enabling the Representation of REDK - Ontology

While our analysis enabled the conceptualization of different knowledge elements, their representation still requires guidelines. For example, while the Service construct may be conceptually understood, its clear representation remains challenging. We use ontology to guide us in representing knowledge elements in a clear and concise way. Ontology, or metaphysics, the philosophy of existence, is the branch of philosophy that deals with modeling the existence of things in the world (Angeles 81). Different ontologies are used within the field of Ontology. The different ontologies take different philosophical positions based on a set of beliefs about the existence of certain entities in reality. In other words, each ontology makes different assumptions about what is perceived to exist in reality and how it behaves. An ontology provides a set of constructs for describing the world.

A specific ontology, widely adapted to IS modeling (e.g. Fettke and Loos 2003, Green and Rosemann 2000, Opdahl and Henderson-Sellers 2002, Wand et al. 1999), is Bunge’s ontology (Bunge 1977, Bunge 1979). This ontology helped analyze concepts of IS (Wand and Weber 1988, Wand and Weber 1995). It was also used to analyze the meaning of constructs used in different conceptual modeling approaches. Specifically, Wand (1989) used ontology to analyze object-oriented concepts. Much work has also been conducted in analyzing modeling grammars (e.g. Maes and Poels 2007, Generoa et al. 2008). Finally, there has also been research in which ontology was used to help translate models between different representation-grammars (e.g. Feng and Wang 2008, Matulevicius et al. 2006).

Bunge’s ontology is well formalized and is "based on the ontological presuppositions of contemporary scientific research, topped with new hypotheses compatible with the
science of the day” (Bunge 1977). Therefore, we use Bunge’s ontology to guide us in
domain knowledge representation. It is important to note that other ontologies could be
utilized for this purpose as well. The use of another ontology might have resulted in the
representation of the same constructs using a different formalism. In the remainder of this
section we describe the core of Bunge’s ontology formalism.

Bunge’s work states the world is composed of things possessing properties. Things are
of two kinds: concrete things and conceptual things. A concrete thing is either simple or
composite, i.e. composed of other things. Attribute can be either intrinsic or mutual.
Intrinsic properties depend on one thing only, while mutual properties depend on two or
more things.

The physical world does not contain concrete things which are bare of properties. An
attribute is a feature assigned by humans to an object. Some of the attributes of a thing
reflect the thing’s properties. Humans recognize properties only through attributes, and a
thing may have a property not known to humans. In other words, properties of things are
materialized through attributes (Wand 1989).

This is formalized via the notion of conceptual schemata (Wand 1989). Things are
modeled via a functional schema in terms of a set of functions that represent the properties
that are important for the purpose of which we model the thing. A functional schema is a
set of attributes along with functions that assign values to those attributes. We call the
functions composing the functional schema state variables. The set of all values assigned
to the functional schema’s attributes by those functions comprises the state of the thing.
Not all combinations of values can be assigned by the state variables to the attributes. Laws
formalize this notion. Law statements are restrictions on the possible values which attribute
functions of the functional schema can be assigned. Laws are fundamental modeling elements, as they contain the knowledge of what a thing can be or not be. Therefore, laws contain dynamic knowledge about things (Wand 1989). The *lawful state space* of a functional schema is the set of attribute states that adhere to all law statements.

When describing dynamic aspects of things, we should consider three fundamental ontological concepts: *changes*, *events* and *interaction*. Every concrete thing has at least two distinct states. A *change* is manifested as a change of the thing’s state. In turn, the change in a thing’s state is an *event*. An event can be represented as an ordered pair `<s1, s2>` where `s1` and `s2` are the states before and after the change respectively (Wand 1989). Two events can be combined into another event. For example the two events `<s1,s2>` and `<s2,s3>` can be combined to `<s1,s3>` (Wand 1989). *Interaction* between objects takes place if the presence of one can affect the states traversed by the other.

However, not all changes of state are lawful. Definitions of the set of transformations from the lawful state space to itself which are possible in the domain are transition laws. Transition laws are defined in terms of a stable state generating change and the transitions’ resulting state.

### 4.3 Representation of REDK

#### 4.3.1 General Outcomes of Ontological Foundations

We now turn to the task of representing the REDK in IS code, guided by ontological principles. In the context of this task we seek representation for concepts which are conceptually clear, but for which representation is not clearly set.
As depicted in figure 8, according to RE two fundamental constructs of the domain are actors and resources. However, Bunge’s ontology does not refer to similar notions. Rather, both actors and resources are mapped to substantial things in ontology. Actors can only be distinguished from resources according to the characteristics they posses. Thus, ontology guides us that although the fundamental elements in most RE approaches are actors and resources, their representation should be very similar as they both map to the same ontological construct of a thing. Therefore, we extended the meta model to include the construct of entity, which maps to a thing in ontology. Resources and Actors are now different kinds of entities in the domain.

An important construct associated with REDK is the State construct. REDK includes knowledge related to states - goals, transitions, and constraints. While RE models use the state constructs extensively, a clear representation of the construct can be found in ontology. As shown previously, the data-based RE literature implies that states are reflected in attributes. This notion is clearly defined in Bunge's ontology. Bunge suggests that a functional schema is a set of attribute functions that assign values at a given time. The set of all values assigned by those functions comprises the state of the thing (Wand 1989). Hence, in order to facilitate the representation of states we extended the meta model to include attributes. With attributes in the model, a state can now be defined by the set of values assigned by the functional schema’s functions.

Other important construct are events, and interaction. While it is clear that roles can be used to facilitate representation of interactions between entities, a representation of this idea is not fully defined. Bunge’s Ontology suggests that interaction is manifested via mutual attributes (Bunge, 1977, p.102). But it only suggests a criterion for recognizing
when interaction has occurred. Namely, if the way attributes of one thing change depends on the presence of another, then the second is said to act on the first (Bunge, 1977, Def. 5.29). Things interact, if and only if one acts upon the other (Bunge, 1977, Def. 5.30). Related to this, in this ontology all possible changes are defined by laws. Moreover, laws relate to attributes of one thing only. Therefore for a thing to act on another thing there must be a mutual attribute between the two. Hence, to facilitate the representation of interaction we augment the model with mutual attributes, which are special types of attributes. We further posit that actor roles must possess mutual attributes.

This leads to another representation issue associated with the service construct. As implied in the current REDK meta-model, RE suggests that an important constraint on services is their enabling and caused states. However, ontology guides us that, in fact, the essence of services is effectively represented by these states. A transition law defines in essence a series of events that can or should take place in the domain. While a transition law does not reflect how things happen, it reflects why and when they happen, and what they are. These are also the elements relevant in the context of domain knowledge. Thus, guided by ontology we represent services using transition laws and assume that transitions are represented by transition laws. A transition law defines whether a state should be changed and if yes, to which. Transition laws are defined in terms of a stable state generating change and the transitions' resulting state. We will further assume that transition enacted by services will reach stable states.

Finally, the way constraints can be represented is also suggested in Bunge’s ontology. As stated in the previous section, Bunge’s ontology defines the notion of laws. Not all combinations of values can be assigned to state variables, and not all transitions are lawful.
Law statements are of two kinds: 1) restrictions on the possible values which attributes of
the functional schema can be assigned with; 2) restriction on the set of possible transitions.
Accordingly, constraints can be represented using state laws and transition laws.

4.3.2 Formalizing Domain Knowledge Constructs

Having outlined the general outcomes of referring to Bunge's ontology for representation, in this section we turn to formally defining each of the REDK constructs using ontological concepts. We take a bottom up approach starting with the definition of the base elements using ontological constructs, and then further integrating the defined elements to help define more complex REDK constructs.

According to Bunge's ontology the real world is made up of things. Related to this, actors and resources are essentially a set of substantial things, as defined in Bunge's ontology. Ontology states that every thing in the domain possesses properties. As such, actors and resources possess properties.

The properties of a thing exist, whether or not humans are aware of them. Humans conceive of things, however, in terms of models of things (Wand et al. 1999). That is, humans relate to conceptual things. In this respect, we distinguish attributes from properties. An attribute is a feature assigned by humans to a conceptual thing. Humans recognize properties only through attributes, and a thing may have a property not known to humans. In other words, properties of things are perceived through attributes (Wand 1989). The value of these attributes at any given time reflects the state of a thing.

Using Bunge's ontology of the definition of state we define:
Knowledge Construct Definition 1:

*State*: The state of any entity in the domain is the vector of values assigned by attribute functions of a thing at a given time.

It is important to note that there are no things without properties (there are no "bare" individuals (Bunge 1977, p. 36). And properties *are always* attached to things ("every property is possessed by some individual or other: there are no properties that fail to be paired to any individuals" (Bunge 1977, p. 62). The properties are of course reflected in attributes. Therefore, we assume that by capturing the states of conceptual entities in the domain (in Requirements Engineering – actors and resources), we capture the state of the domain.

**Corollary 1**

Domain state: The state of the modeled domain is reflected in the states of all entities in the domain.

Recall that according to Bunge attributes can be one of two kinds. They can either be intrinsic or mutual. Recall that intrinsic attributes are dependent only on a single entity – the entity that posses the attribute. An example of an intrinsic attribute is age or height. Also recall that mutual attributes exist between two or more entities. An example of a mutual attribute is the position of an employee in a company. The distinction between those
two types of attributes helps us formally distinguish between actors and resources using ontological constructs.

**Knowledge Construct Definition 2:**

**Actor:** An actor is a representation of a thing in the domain, which has at least one mutual attribute it can change.

**Knowledge Construct Definition 3:**

**Resource:** A resource is a representation of a thing in the domain which can not change any of its mutual attributes.

Bunge (1977) defines a "kind" as a set of things having a set of properties. A "natural kind" is a set of things possessing lawfully related properties. While it is not explicitly indicated in Bunge's ontology, we will assume here that laws defining a kind include transition laws. While roles are not naturally defined in ontology, using the notion of kinds and natural kinds, ontology helps define them.

Uschold et al. (1998) identify two kinds of roles — those that define a general relation, and actor roles, which indicate interactions among actors and entail some activity or cognition for one of them. Extending from Wand et al. (1999) ontological definition of kinds, we can ontologically represent general and active roles.

Since Bunge's ontology posits that mutual attributes must be possessed by a thing in order to entail activity, an actor role must possess at least one mutual attribute. Further, since a role encompasses how a thing is represented, the ontological meaning is that a role
has attributes (reflecting the way humans perceive a thing). Thus, a role in general is an ontological kind, and an actor role is a kind which incorporates at least one mutual attribute it can change:

**Knowledge Construct Definition 4:**

**Role:** A kind of thing in the domain.

**Knowledge Construct Definition 5:**

**Active Role:** A natural kind with laws in the domain, possessing at least one mutual attribute and at least one transition law that modifies a mutual attribute.

Recall, that the definition of an actor above was related to this definition of active role. That is, a conceptual thing can play an active role only if it affects at least one mutual attribute. On the other hand, a conceptual thing possesses a mutual attribute only if it is an Actor. Using this relation between actors, active roles, and mutual attributes, we can alternatively define an actor as.

**Corollary 2**

Actor: An actor is a conceptual thing in the domain which plays at least one Active Role

Note that this definition enables an entity to have both active roles and regular (passive roles). For example, a computer may have a passive role when the role only incorporates intrinsic attributes and when it cannot change any mutual attribute (for example the role: a
The concept of a state of things enables us to define goals as well. The major difference is that an actor is active while a resource is not. Another difference lies in the fact that an actor is goal driven while a resource is not. That is, an actor typically has a set of state which the actor desires to achieve. While there is no imposition that all actors must have goals, the goal-oriented approach suggests that things are done in order to satisfy goals. An Actor’s goal is a set of values the actors’ attributes is desired to have.

*Knowledge Construct Definition 6:*

*Goal:* a goal is a set of states in the domain.

When coming to represent a service ontologically, we need to consider the ontological constructs used to represent dynamic aspects. In this respect, it is noticed that transitions and transition laws are the sole representations of dynamics. Ontologically, events change the state of a thing and a transformation is a mapping from a domain of states to a codomain of states (Wand and Weber 1995). Transition law defines the essence of a series of events that can or should take place in the domain. Transition laws are defined in terms of unstable state (representing the need for the service to be provided) and, following the stability assumption, the stable state arrived at as a result of the service execution. Since a
service is provided by actors only, transition laws are defined for active roles only – the roles which define actors.

Knowledge Construct Definition 7:

Service: a service is an ability to enact a transition law from a domain of states to a codomain of states.

It is interesting to note that in effect this definition enabled us to link the three constructs of event, service and transition in one definition of a service. A service is in effect a transition, which is specified by an unstable state (precondition), an event and a resulting stable state (the post condition). This definition aligns well with Uschold et al’s (1998) definition of the three.

Finally we turn to the definition of constraints. Recall that Uschold et al. (1998) define three types of constraints – precondition, postcondition and activity constraints. Ontology guides us that these can be represented in terms of lawful states and unlawful states. That is, the constraint construct can be represented in terms of state laws and transformation laws (Wand and Weber 1999)

Knowledge Construct Definition 8:

Constraint: A constraint is a condition defining lawful states or lawful transitions.

Table 3 summarized the mapping between REDK constructs and ontological constructs. In the second column of Table 3 we show how the REDK constructs can be mapped to
ontological constructs. Since REDK is only a subset of what may be full domain information, REDK constitutes a more restrictive set of constructs. We explicitly indicate this restriction in the third column of the table. The resultant way to represent REDK constructs ontologically is provided in the fourth column.

Table 3. Ontological Representation of REDK Constructs.

<table>
<thead>
<tr>
<th>REDK Construct</th>
<th>Mapping to Ontological Construct</th>
<th>Constraints on Ontological Construct to reflect RE meaning</th>
<th>Resulting REDK Construct Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor</td>
<td>A thing</td>
<td>Affects at least one mutual attribute</td>
<td>A thing affecting at least one mutual attribute</td>
</tr>
<tr>
<td>Resource</td>
<td>A thing</td>
<td>Does not change any mutual attribute</td>
<td>A thing which does not change any mutual attribute</td>
</tr>
<tr>
<td>Active Role</td>
<td>A natural kind</td>
<td>Can change at least one mutual attribute</td>
<td>A natural kind, which can change at least one mutual attribute</td>
</tr>
<tr>
<td>Role</td>
<td>A kind</td>
<td>Does not necessarily affect a mutual attribute</td>
<td>A kind, which does not necessarily affect a mutual attribute</td>
</tr>
<tr>
<td>State</td>
<td>State (The vector of values assigned by attribute functions of a thing)</td>
<td>-</td>
<td>The vector of values assigned by property functions of a thing</td>
</tr>
<tr>
<td>Goal</td>
<td>A State</td>
<td>A desired vector of values</td>
<td>A desired set of values to be assigned by property functions of a thing</td>
</tr>
<tr>
<td>Service /Transition</td>
<td>Laws (Transition from Unstable to Stable states)</td>
<td>A definition of a transition law</td>
<td>A definition of a transition law from unstable state to a stable state</td>
</tr>
<tr>
<td>Constraints</td>
<td>State Laws and Transition Laws</td>
<td>Conditions for lawful transformations and states</td>
<td>The conditions defining lawful transformations and states</td>
</tr>
</tbody>
</table>

The refined model incorporating ontological representation constructs is reflected in figure 10. Note that while resource possesses only intrinsic attributes and only actors may incorporate both intrinsic and mutual attributes, this is not indicated in the figure to avoid
excessive number of lines. As well, for clarity purposes, although lawful transition and transition law map to the same ontological construct of transition, we present the two separately in the figure. Finally, while mutual attributes are associated with a set of entities, for clarity this is not depicted. In the figure, “manifests” (represented by a dashed arrow) indicates that the manifesting construct is used to represent another construct.

Figure 10: Representation of requirements engineering domain knowledge

We also provide explicitly the relationship between the constructs as depicted in the meta model of figure 10. These relationships are found in table 4
### Table 4. REDK Relationships

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Representation in figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>The domain is composed of entities</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>An Entity may be a composition or generalization of other Entities</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>A Resource is a type of Entity</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>An Actor is a type of Entity</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>Relationship</td>
<td>Representation in figure</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td><strong>An Actor plays Active Roles</strong></td>
<td><img src="image1.png" alt="Diagram" /></td>
</tr>
<tr>
<td><strong>Active Roles interact with Roles</strong></td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
<tr>
<td><strong>Active Role is a type of Role</strong></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td><strong>Active Roles can perform Services</strong></td>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Relationship</td>
<td>Representation in figure</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>Services change Attributes</td>
<td><img src="image" alt="Services change Attributes Diagram" /></td>
</tr>
<tr>
<td>An Entity is in a State</td>
<td><img src="image" alt="An Entity is in a State Diagram" /></td>
</tr>
<tr>
<td>A State is composed of Attributes</td>
<td><img src="image" alt="A State is composed of Attributes Diagram" /></td>
</tr>
<tr>
<td>Intrinsic Attribute is an attribute</td>
<td><img src="image" alt="Intrinsic Attribute is an attribute Diagram" /></td>
</tr>
<tr>
<td>Relationship</td>
<td>Representation in figure</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Mutual Attribute is an attribute</td>
<td><img src="#" alt="Diagram" /></td>
</tr>
<tr>
<td>Lawful state is a State</td>
<td><img src="#" alt="Diagram" /></td>
</tr>
<tr>
<td>Beginning state is a State</td>
<td><img src="#" alt="Diagram" /></td>
</tr>
<tr>
<td>Resulting state is a State</td>
<td><img src="#" alt="Diagram" /></td>
</tr>
<tr>
<td>Relationship</td>
<td>Representation in figure</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>A Lawful Transition starts in a beginning state</strong></td>
<td><img src="image" alt="Lawful Transition starts in Beginning State Diagram" /></td>
</tr>
<tr>
<td><strong>A Lawful Transition ends in a resulting state</strong></td>
<td><img src="image" alt="Lawful Transition ends in Resulting State Diagram" /></td>
</tr>
<tr>
<td><strong>A Transition Law is composed of one stable and one unstable state</strong></td>
<td><img src="image" alt="Transition Law Diagram" /></td>
</tr>
<tr>
<td><strong>A Stable State is a State</strong></td>
<td><img src="image" alt="Stable State Diagram" /></td>
</tr>
</tbody>
</table>
4.4 Using REDK Representation Constructs to Represent Domain Knowledge Elements

In this section we show the use of REDK constructs to model domain knowledge gathered in RE. We adopt the course-exam management case study described in Tropos (Fuxman et al. 2004) and show how the constructs enable us to model domain knowledge gathered during the development of the system in the case. Moreover, the constructs focus on domain knowledge alone, elements that cannot be modeled using the REDK constructs do not relate to domain knowledge.
The Tropos case study (Fuxman et al. 2004) is about a system to support teachers in running and marking exams. We refer to i* models used in the case and show how REDK models can reflect explicit and implicit domain knowledge found in the original i* models. The models provide an informal graphical description of the organizational setting obtained during requirement engineering.

4.4.1 The Tropos Case

In order to understand the organizational domain and elicit requirements, tools and techniques such as interviews, questionnaires, and analysis of competing organizations takes place. At the early phases stakeholders are identified along with their intentions. The i* modeling process emphasizes the need to understand and model the strategic aspects underlying the organization in which the system will eventually operate. According to the philosophy underlying i*, by understanding strategic aspects in the environment motivations for the system can be better identified. Figure 11 depicts the high-level i* model of the Tropos case study (Fuxman et al. 2004).

Figure 11: Strategic dependency i* model of the Tropos case (Adapted from Fuxman et al. 2004)
Table 5. The Tropos Case - Constructs, Meaning, and Representation.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Meaning in Tropos (Bresciani et al. 2004)</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>Actors’ strategic interests.</td>
<td></td>
<td>Passing a course is a goal</td>
</tr>
<tr>
<td>Softgoal</td>
<td>A goal without clear criteria for its accomplishment</td>
<td>![Symbol]</td>
<td>Integrity is a soft goal</td>
</tr>
<tr>
<td>Actor</td>
<td>Physical, social or software agent as well as a role or position</td>
<td>![Symbol]</td>
<td>Student is an actor</td>
</tr>
<tr>
<td>Resource</td>
<td>Physical or an informational entity</td>
<td>![Symbol]</td>
<td>Lectures are resources</td>
</tr>
<tr>
<td>Plan\Task</td>
<td>Represents, at an abstract level, a way of doing something</td>
<td>![Symbol]</td>
<td>Giving exams is an activity</td>
</tr>
<tr>
<td>Dependency</td>
<td>Indicates that one actor depends, for some reason, on the other in order to attain some goal, execute some plan, or deliver a resource</td>
<td>![Diagram]</td>
<td>Student depends on teacher to get mark</td>
</tr>
</tbody>
</table>

The five domain constructs covered in this model are actors, goals, tasks, dependencies, and resources. Goals can either have, or not have, clear-cut criteria as to when they are achieved. As discussed in the previous sections, if the criteria are not clear-cut the goal is termed in i* a soft-goal. In i*, the first type is modeled using a rounded rectangle, the latter is modeled using a curvilinear shape.

According to the Tropos case discussed, (figure 11), two types of actors exist in the domain. These are students and teachers, modeled using a circle. The goals in this case are the student’s objective to pass a course, Pass[Course]; the teacher’s expectations of
students to be honest (Honesty); and the teacher has the goal to attract students (AttractStudents). The latter two goals are soft goals since there are no clear criteria to determine their achievement.

A task in i* is modeled with a hexagon. In i*, a task specifies a particular course of action that produces a desired effect. In our specific Tropos case, the element Give[Exam] represents a task. Finally, resources are modeled in i* with a rectangle. In figure 11, three resources are modeled. Students wait for the lectures of the course (Lectures[Course]) and for a mark for an exam (Mark[Exam]). The teacher waits for an answer to an exam (Answer[Exam]).

In i* a boundary delimits intentional elements (goals tasks and resources) that are internal to each actor. Intentional elements outside the boundaries correspond to goals, tasks and resources whose responsibility is delegated from one actor to another. For example, the student depends on the teacher for the marking of the exams. In the figure, dependency links are used to represent these inter-actor relationships.

Under Tropos, we can further zoom into the actor and show how high level intentional elements are operationalized and broken down using a strategic dependency model (figure 12). This provides more details of the i* actors in the domain and their relationships. In our case study, the student and teacher are analyzed. In this analysis intentional elements are refined and operationalized using means-end, contribution and decomposition links. Means-end represent alternative ways to achieve a goal. For instance, the case model, depicted in figure 12, shows that in order to pass a course (Pass[Course]), a student can pass all the exams of the course (Pass[Exam]), or can do a research project for the course
(DoResearchProject[Course]). Decomposition links define a refinement for a task. For instance, if a student wants to pass an exam (Pass[Exam]), the student needs to attend the exam and get a passing mark (GetPassingMark[Exam]). A contribution link describes the impact that one element has on another element. That is, the impact can be either negative or positive. In the Tropos case, FairMarking[Exam] has a positive impact on Pass[Exam], since a fair marking makes it easier for the student to judge when he is ready to take the exam.
Table 6. The Tropos Case – Means End Analysis Constructs, Meaning, and Representation.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Meaning in Tropos (Bresciani et al. 2004)</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>Refinement for a task. (AND decomposition of a task)</td>
<td></td>
<td>Run Exam → Give Instructions Get Answers</td>
</tr>
<tr>
<td>link</td>
<td></td>
<td></td>
<td>To run an exam give instructions and get answers.</td>
</tr>
<tr>
<td>Contribution</td>
<td>The impact one element has on another element. The contribution can be positive or negative (a negative or positive can be attached; if non is attached the contribution is positive)</td>
<td></td>
<td>Integrity → Know answer</td>
</tr>
<tr>
<td>link</td>
<td></td>
<td></td>
<td>Knowing answers contributes to integrity</td>
</tr>
<tr>
<td>Means-End</td>
<td>OR decomposition of a tasks</td>
<td></td>
<td>Pass course → Do project Take Exam</td>
</tr>
<tr>
<td>link</td>
<td></td>
<td></td>
<td>To pass a course do a project or take an exam.</td>
</tr>
<tr>
<td>Dependency</td>
<td>Indication of interactor relations (direction is from the dependant to the dependee)</td>
<td></td>
<td>Study → Lectures</td>
</tr>
<tr>
<td>link</td>
<td></td>
<td></td>
<td>Studying is dependent on lectures.</td>
</tr>
</tbody>
</table>
We now show how these elements relate to our REDK constructs. An REDK based model would differ from an i* model for two reasons. First, i* is employed to facilitate system requirements elicitation. As such, both elements to be changed by the system and unchanged domain elements can be incorporated in an i* model. However, REDK attempts to model domain knowledge that does not depend on the incorporation of the system, thus excluding as-is system-based knowledge from the model.
Second, since REDK incorporates constructs used throughout the RE literature, some meta model constructs not supported in i*, may be relevant to support a clearer representation of the domain. For example, roles (active or general) are part of REDK, but not part of the i* meta model. Table 6 shows the mapping between Tropos and REDK.

<table>
<thead>
<tr>
<th>Tropos</th>
<th>REDK</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor</td>
<td>Actor and (Active) Role</td>
<td>Students are roles in REDK Students are actors in Tropos</td>
</tr>
<tr>
<td>Resource</td>
<td>Resource and (Passive) Role</td>
<td>Exam is a resource in Tropos Exam is a passive role in REDK</td>
</tr>
<tr>
<td>Dependency (of different kinds: task and goals)</td>
<td>Mutual Attribute</td>
<td>In Tropos teacher depends on Answers to exam by students In REDK students and teachers share the mutual attribute answers in exam.</td>
</tr>
<tr>
<td>Task</td>
<td>Service</td>
<td>Taking an exam</td>
</tr>
<tr>
<td>Task Refinement</td>
<td>Stable/Unstable states</td>
<td>Different levels of analysis. Tropos concentrates on the details of processes. REDK concentrates on their causes and effects in the environment</td>
</tr>
<tr>
<td>Goal</td>
<td>Goal</td>
<td>Student have a goal of passing a course</td>
</tr>
</tbody>
</table>

As can be inferred from the table, some domain elements are well represented, and some are not well represented in i* (such as Attributes and Constraints). Therefore, the domain elements can be found implicitly or explicitly in the i* diagram of figure 12.

According to this, we model the domain elements extracted during the Tropos processes using REDK constructs. We use the i* grammar, for the constructs that have a
one to one mapping between REDK and i* (i.e. when constructs have direct representation in REDK and i*). For those REDK constructs which do not have a direct representation in i*, we define the notation in table 8. These include general roles (a double circle); active roles (a circle with a star); mutual attributes (hexagon); and services (rectangle with stable\unstable state pairs).

The general rules we applied when converting the i* diagram to REDK based diagram are as follows:

1. Clearly indicate whether an i* Actor is an Actor or an Active Role: i* refers to actors as active roles—this makes the i* elements unclear when acquiring domain knowledge. For example, when modeling a corporation, i* would model a specific person in the corporation using similar constructs as “Employee”. That is, in i* they are both actors. In REDK there are different constructs to represent the two. The former is an actor and the latter is an active role.

2. Clearly indicate whether an i* Resource is a Resource or a (Passive) Role: i* refers to resources as general roles—this makes the i* elements unclear when acquiring domain knowledge. Moreover, it makes it unclear that the same entity may acquire and lose roles, and may play several roles simultaneously. For example, a question (passive) role may also play the (passive) role of an exam. In i* modeling an entity playing both roles is not possible.

   Moreover, without prior domain knowledge it would be hard to understand if an element is a resource or just a role of the resource. For example, the resource “room 123” and the role “meeting room” are both modeled as resources in i*. In
REDK there are different constructs to represent the two. The former is a resource and the latter is a (passive) role.

3. Indicate attributes for entities: in i* attributes of entities are not modeled. For example, one would not find attributes such as "Name", "Weight" etc modeled in i*. Those attributes that are only inferred in the i* model should be explicitly represented in an REDK model. For example, while the above attributes are not modeled one may find tasks associated with the attributes, such as "change name" or "loose weight" etc. i.e. the existence of those attributes can be inferred.

4. Make interaction points explicit by including mutual attributes: in i* dependencies are modeled but how interaction takes place is not shown. Therefore, while the creation of the model required underlying domain knowledge about interaction, interactions are not modeled explicitly. With REDK constructs, interaction can be modeled using mutual attributes. Therefore, whenever a dependency is noticed in an i* model, a mutual attribute is modeled in the REDK model.

5. Exclude from the REDK diagram elements associated with processes that might change with the introduction of the developed Information System: In i* tasks are decomposed and refined. However, these refinements in effect provide details of processes that the introduction of the information system might change. In REDK we model only the stable and unstable states in the environment. Tasks are performed to transform an unstable state to a state. The way they are performed is not shown.

The outcome of this modeling process is depicted in figure 13. By examining the figure, the enhancements in modeling domain knowledge using REDK constructs becomes
quite clear. First, referring to guideline 1, using the ability of REDK to model domain knowledge commonly referred to in RE, our vocabulary is extended to include active roles. With this, the two domain elements modeled as actors in i*, namely Student and Teacher, are modeled as roles in our diagram. Entities in the domain may acquire and lose these roles dynamically and maybe even play them simultaneously. Next, referring to guidelines 2 and 3, we note that resources which did not have general roles, and therefore did not have any attributes in the i* model, now incorporate attributes. Specifically, the roles Course, Lecture, Exam, and Report, are now explicitly represented in our REDK diagram using (passive) roles. All resources now possess attributes.

Table 8. Additional REDK construct Representation

<table>
<thead>
<tr>
<th>Construct</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active role</td>
<td><img src="image" alt="Active role" /></td>
</tr>
<tr>
<td>(Passive) role</td>
<td><img src="image" alt="Passive role" /></td>
</tr>
<tr>
<td>Attribute</td>
<td><img src="image" alt="Attribute" /></td>
</tr>
<tr>
<td>Mutual Attribute</td>
<td><img src="image" alt="Mutual Attribute" /></td>
</tr>
<tr>
<td>Service</td>
<td><img src="image" alt="Service" /></td>
</tr>
<tr>
<td>State definition includes attribute (can be mutual or intrinsic)</td>
<td><img src="image" alt="State definition" /></td>
</tr>
<tr>
<td>Constraint</td>
<td><img src="image" alt="Constraint" /></td>
</tr>
</tbody>
</table>

115
Specifically, attributes, which had to be identified when constructing the i* diagram, but were not explicitly included, are now represented. For the Exam resource, these
attributes are *Instruction*, *Mark*, *Answer*, and *Marking Scheme*. Knowledge about these attributes had to be accumulated when constructing the $i^*$ diagram in figure 12 since tasks, resources, and goals are associated with them (such as *mark exam* task, *fair marking* goal, etc.), however they are not explicitly represented. For example, we can extract that *Answer* is an attribute of *Exam* since figure 12 includes a task *Give Answer on Exam*.

To this end, referring to guideline 4, mutual attributes are explicitly represented in our model as well. Once again, knowledge about their existence had to be established while constructing the $i^*$ model. However, they can only implicitly be found in the $i^*$ model, and sometimes they may have even been left out. For example, the *Answer* attribute is a mutual attribute of *teacher*, *exam*, and *student*. Similarly a query about an exam refers to a mutual attribute between *teacher* and *student*, since the student states the query and the teacher relates to it. Their existence is evident when considering dependencies in the figure. In the $i^*$ diagram different types of dependencies are modeled, without clear modeling of the way they can take place. That is, in order for a dependency to exist, some mutual attribute must exist as well. This link is often missed in $i^*$ models. For example, in the $i^*$ diagram of figure 12 we have *integrity* as a softgoal of a student. We can also notice that a softgoal of the teacher is to have *honesty*, which is dependent on students' integrity. Further, the $i^*$ model reflects that honesty affects the teacher in giving an exam. However, these goals are not associated with anything else. With some analysis, we can understand that since honesty affects giving an exam in a course, it is associated with a course. We further understand that this goal is dependent on a mutual attribute, shared by course, student, and teacher, since the student affects it with his/her integrity softgoal and the teacher is influenced by it in giving the exam. Hence, mutual attribute *course integrity* is shard
between course, student, and teacher, as depicted in the REDK model. This mutual attribute affects the honesty softgoal of the teacher, and is related to the integrity softgoal of the student.

Referring to guideline 5, in the diagram we note that elements not related to domain knowledge that will not remain valid after system incorporation, are no longer found. Specifically, the ways of going about in executing processes are not modeled. This is since the way things are done is likely to change once the system is incorporated. I.e. knowledge about those becomes knowledge about the system, and not about the domain, as the system is incorporated into the business. In fact, the incorporation of tasks into the i* diagram has potential for causing confusion as goals may be confused with tasks. For example, in the case’s diagram PassExam is modeled as a task. However, since PassExam is a state and not a transition between states, it is in fact a goal. Since this goal would typically be accomplished as part of the exam-taking process, it may be possible to model it as part of the process. However, modeling it as a goal would be more accurate.

Since REDK models only domain knowledge, tasks are not modeled; only services are. Since tasks are not modeled, but only the service outcome is modeled, things that need to be done to fulfill a service (tasks) are not confused with goals. In our diagram PassExam is a goal. Furthermore, since REDK models resources’ roles and their attributes explicitly we can further see that PassExam is a function of the exam Mark attribute modeled. Similarly, other task refinements are also excluded from the REDK model. However the stable and unstable states, leading to and resulting from services, are modeled. For example, we see that a student would take an exam if he/she did not complete a report and exam time has arrived. We also see that taking exam process is complete when the exam is filled. The
diagram also reflects the fact that this process may affect the *Pass Exam* goal. Similarly we can find stable/unstable pairs for other services performed by students and teachers. These include *TakeExam*, *DoResearchProject*, *StudyCourse*, *PrepareExam*, *MakeExam*, and *Exam Discussions*. Again, while the conditions for their initiation and completion are not reflected in the i* diagram of figure 12, they must have been understood upon creation of the diagram. These constructs are part of REDK.

In this diagram we explicitly see some domain knowledge which was only in the mind of the modeler in i*. Explicit representation of domain knowledge can enable additional people to use the model both to elicit requirements and to learn about the domain.

Having REDK constructs defined and having suggested how they can be represented, in the next two sections we discuss some of the possible usages of REDK, and how it may be incorporated in system development stages.
5. Framework for Supporting Development with REDK

5.1 General

This section demonstrates how a systems analyst can use REDK to document domain knowledge, independent of which RE modelling approach they choose to use. For this purpose, we provide an example where two common RE approaches: the data-based approach and the goal-oriented approach. We show that while the two approaches facilitate obtaining similar requirements, they differ in the way the analyst may deduce these requirements. Importantly in our context, is that the approaches differ in the domain knowledge elements they document. We show how, using the REDK constructs, it is possible to document all domain knowledge in a single diagram. We then show briefly that, if an actual logical design is provided, it can be checked against the REDK-based model. Finally, we point out the differences between the REDK representation and the original data-based and goal-based models.

5.2 Development Framework Illustration

To illustrate the benefits of REDK use to systems analysts, we describe the development and requirements elicitation of a Customer Relationship Management System (CRM). In this example, the ability to use REDK constructs to document domain knowledge gathered during RE is demonstrated. Further, we show how REDK
documentation provides the grounds to relate possible implementation design to domain knowledge obtained.

In this section we do not relate to a specific system, but rather describe some functionalities which are typically supported to some extent in typical CRM systems. In the discussed example, an organization wishes to incorporate a new system which will help manage incoming queries and requests from customers. Using this system, the handling of customer requests will be partially automated.

In order to understand the organizational domain and elicit requirements, different RE approaches can be employed - tools such as interviews, questionnaires, and analysis of competing organizations takes place. When considering a goal-oriented RE approach, during the early phases stakeholders are identified along with their intentions. Considering i*, the strategic dependency diagram depicted in figure 14 is used to reflect this initial analysis. This figure shows that the basic understanding of stakeholders in the organization is that the domain consists of customers, potential customers, managers, and employees. Customers are associated with the goal of using the company’s products effectively; the managers’ goal is to retain and increase company profits; employees’ major goal is to increase their income; and potential customers try to gain product information. Softgoals are shown in the figure as well. Employees would like to have satisfied customers since handling such customers is easier. However, it can be seen that they depend on the intended system’s functionality to achieve this softgoal. This is modeled in the diagram via the dependency arrow from Employees to the system. Managers would like to increase customer service efficiency, which is dependent on the intended system as well. Finally, customers
would like to feel that they have spent their money well when they have purchased the product.

Table 9. Goals and Softgoals in Development Example.

<table>
<thead>
<tr>
<th>Role</th>
<th>Goals</th>
<th>Softgoals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management</td>
<td>Retain customers and</td>
<td>Increase Customer Service</td>
</tr>
<tr>
<td></td>
<td>Increase Company Profit</td>
<td>Quality</td>
</tr>
<tr>
<td>Employee</td>
<td>Increase Salary</td>
<td>Have satisfied customers</td>
</tr>
<tr>
<td>Customer</td>
<td>Use Product Effectively</td>
<td>Money Well Spent</td>
</tr>
<tr>
<td>Potential Customer</td>
<td>Get Product Info</td>
<td>None</td>
</tr>
</tbody>
</table>

Based on this initial analysis, goals are further analyzed and decomposed. The assumed process at this stage is reflected in the goal diagram of figure 15. For clarity purposes, since all arrows in the diagram are contribution links, they are depicted as solid arrows rather than dashed ones. As previously indicated, contribution arrows generally indicate a positive contribution, however when notated with a negative sign they indicate a negative contribution.

The analysis depicted in figure 15 reveals that the most basic domain actors are the potential customers. Their goal of obtaining information about products can be achieved either by browsing the company’s website or by calling the company’s customer service lines. Customer goals are more complex. In order to achieve the goal *use product effectively* a customer would typically want to achieve the subgoals of *activating product with minimal delay*, and *use features needed*. Along similar lines, the softgoal *money well
spent is decomposed into the goals *get better support than offered by competing companies* and *get features I need.*

Managers' goal of retaining and increasing company profit is decomposed to the goals of retaining valuable customers, increasing the business conducted with valuable customers, and increasing the number of customers in general. The analysis also reveals

<table>
<thead>
<tr>
<th>Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
</tr>
<tr>
<td>Softgoal</td>
</tr>
<tr>
<td>Actor</td>
</tr>
<tr>
<td>Dependency</td>
</tr>
</tbody>
</table>

Figure 14: Strategic Dependency diagram of the implementing organization domain
that if managers work through customers quickly the number of customers in general may increase, but retaining valuable customers, which deserve special attention, may be negatively affected. Finally, managers are found to be more effective in achieving the goals of retaining valuable customers and increasing their business since managers have extensive knowledge of the products being sold, a property required for achieving these goals.

On a parallel line, employees are analyzed. It is found by the system analyst that, due to the compensation method employed by the company, employees’ goal of increasing salary would be achieved by responding to as many customers as possible each day. Since this is employees' important goal, in order to achieve this goal employees try to give simple explanations, and therefore their expertise is primarily related to the basics of the product. Giving simple answers will generally make customers feel more relaxed, a softgoal of the employees (see figure 15). However this is not necessarily the case. Sophisticated customers, who are valuable as they often have extensive business with the company, may demand thorough explanations. Employees’ lack of knowledge in handling more complicated questions may upset such customers.

We now assume the analysis revealed an important set of actors in the domain, which is recognized only implicitly in the organization’s stakeholders’ minds. Namely, customers which have large accounts in the company, valuable customers, surface in this analysis. They are very different from regular customers. They are not likely to appreciate being treated by regular employees with limited product knowledge, and they should not be passed through quickly. Another notion revealed is that potential customers are not very different from customers — they would usually need only simple answers, and treating them
quickly would contribute to the managers’ goals of CRM efficiency and increased company profit.

Furthermore, assume that the refinement of goals reveals that managers are well equipped to work with valuable customers, while employees are well equipped to work with regular customers. The main reason is managers’ major goal is to increase the valuable customer base. In order to achieve this goal managers provide thorough information and work carefully with the customers. This has positive effects on increasing valuable customer base, but negative effects in increasing regular customer base. On the other hand, regular employees tend to minimize the time they spend with customers to achieve their compensation goals. This aligns well with handling regular customers, but conflicts with handling valuable customers. Hence, it appears that a requirement of the system is that employees would be assigned by the system to handling regular and potential customers while managers will be assigned with handling valuable customers.

However, the analysis reveals that an important tool used by employees and managers in handling customers should be considered as well. Managers and employees routinely give discounts on different occasions to valuable customers. Each valuable customer is associated with an attribute indicating the maximal discount the customer may be granted, and if dissatisfaction is expressed by the valuable customer, a discount up to that amount can be granted. In this regard, it appears that managers are well equipped in handling any such customer effectively. Managers would carefully consider the discount given to a customer, and would suggest different discounts according to the value of the customers and the case at hand. On the other hand, employees tend to provide the largest discount possible in order to quickly satisfy the customer and work through to another customer.
This may work well if the customer is indeed a valuable customer. However if the customer is not valuable, large discounts will result in lost profits to the company. Hence, according to this criterion it appears that employees should only handle valuable customers. In order to overcome this conflict, it is realized by the analyst that regular employees should not be allowed to give any discount. This way, employees will still be able to provide effective handling of regular customers, but will not be effective in handling valuable customers. Now employees' abilities, according to product expertise and according to discounting abilities, align with handling specific type of customers. Regular employees are best fit to handle regular customers, both according to discounting considerations and according to product expertise considerations. Managers are fit to handle valuable customers, both according to discounting considerations and according to product expertise considerations (and of course, the expectations of these customers as indicated above).

Based on this analysis, the final refinement of the diagram is provided in figure 16, and the following system requirements are elicited:

**Table 10. System Requirements.**

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Managers should be assigned with handling only valuable customers</td>
</tr>
<tr>
<td>2.</td>
<td>Employees should be assigned with handling only regular customers</td>
</tr>
<tr>
<td>3.</td>
<td>Only managers give discounts</td>
</tr>
<tr>
<td>4.</td>
<td>Valuable customers should be handled only by managers.</td>
</tr>
<tr>
<td>5.</td>
<td>The system should be configurable as to the maximal number of customers an employee can handle.</td>
</tr>
</tbody>
</table>
Figure 15: Strategic Rational diagram of the implementing organization's domain (legend see table 5, page 107)
Alternatively, if considering an analysis taken under the data-based approach, different elements of the domain would have been modeled. The first assumed diagram created in the process is depicted in figure 17. In this model, elements such as actors are goals are not present, and emphasize is put on data-based constructs.

Initially the domain is viewed as consisting of potential customer and is not viewed to have valuable customers. However, this view indicates that both managers and regular employees need to handle customers in different cases. Managers need to handle customers
which need extensive information, and employees can handle them when basic info is needed.

Figure 17: Class Diagram - Data based analysis of the organization's domain

This leads to a deeper analysis regarding when extensive information is needed, which indicates that only valuable customers require such information. Since only valuable customers require extensive information, this class of customers is defined. Now valuable customers are handled by managers and regular customers are handled by employees, since they do not include the service “GetExtensiveInfo” which requires managers’ help. This situation leads to the realization that the class of Potential Customer and Customer are identical, leading to the unification of the two classes and thus eliminating the class of Potential Customer. The new class diagram is thus depicted in figure 18. Note, that there might be other interactions that will keep the distinction between potential customer and customer; however these are outside the domain of analysis.
Hence, once again, although using a different RE approach, the requirements depicted in table 10 are arrived at. The analysis revealed that the domain is found to consist of customers and valuable customers - a special kind of customers eligible for discounts (limited by a Max Eligible Discount). As well, the domain consists of employees and a special kind of employees - managers, who can allow discounts (limited by a Max Discount). Finally, while employees serve regular customers, managers also serve valuable customers. The model also reflects a service provided by all employees (managers and regular) – HandleRequest - by which employees process a customer request. In the process of analyzing the domain, the conceptual model also documents the context of services. Conceptualizations of HandleRequest is shown in figure 19. This figure depicts knowledge about conditions for providing services, and the state of entities upon service completion is made clear. A phone call triggers the Handle Request action. Processing is complete when the customer offline state is reached.
Figure 19: Conceptual model of Handle Request service

Note however, that using REDK constructs we can model of the domain knowledge described as elicited during the RE processes. Figure 20 encompasses the data-based and goal-based domain knowledge, gathered in these examples, combined.
Figure 20: REDK of the CRM Example

Figure 20 incorporates the i* process based domain knowledge. The figure was created by applying similar rules to those provided in section 4. Namely:

The general rules we applied when converting the i* diagram to REDK based diagram are as follows:

1. Clearly indicate whether an i* Actor is an Actor or an Active Role. A Role is modeled when knowledge about the element relates to interaction in the domain. In the case, the different actors defined are in fact all active roles – Manager, Employee, Regular Customer, and Valuable Customer are all active roles. This is depicted in the figure.
2. Clearly indicate whether an i* Resource is a Resource or a (Passive) Role:

A passive role is modeled when knowledge about the element relates to the way something is used or consumed. In our case, the passive role product is modeled in REDK.

3. Entities should have attributes. In the case, many attributes are now explicitly modeled in the diagram for different roles. These include Transaction, Discount, and Answer. These are elicited from analyzing the goals of entities, and what affect them. For example, we know Transaction mutual property exists between managers and customers because we know managers are concerned with the transactions made by valuable customers, as it affects their goal of increasing company profit.

4. Make interaction points explicit by including mutual attributes: Mutual attributes are made explicit to enable interaction. These include Phone Conversation, Discount, and Transaction. Extending from the example in guideline 3 above, we know that Transaction is an interaction point between valuable customers and managers.

5. Exclude from the REDK diagram elements associated with processes that are subject to change with the developed Information System. In the REDK diagram we only include services. Services are modeled using Stable and Unstable states. In the case, these include Handle Request, Consider Discount, and Activate.

Note that figure 20 also incorporates domain knowledge found using the data-based approach (of course some of this knowledge overlaps with the knowledge gathered in the goal based approach). For depicting data-based domain knowledge, the following rules were applied:
1. An attribute which is modeled at the conceptual model, should be modeled in the REDK diagram as well. In the example, we have the attributes Address and Name in Customer and Valuable Customer.

2. Model the stable and unstable states associated with services: Hence we have the services GetBasicInfo and GetExtensiveInfo modeled using stable and unstable states.

3. For each attribute modeled, make sure it is associated with some service or constraint, and model the constraint: In the example, Number of requests handled is modeled as associated with the constraint – has to be greater than 45.

Hence we see that with REDK, we can model all domain knowledge gathered using different RE approaches. Figure 20, explicitly shows all the domain knowledge gathered during the process of eliciting requirements.

We now further use this example to demonstrate how REDK based modeled domain knowledge can serve as a basis to compare domain knowledge to implementation design. That is, by modeling REDK, one can examine how accurately all domain knowledge is represented in implementation code.

The analysis phase leads the way to the design phase of the information system. We examine a possible implementation under conventional development procedures (leading to domain knowledge embedded in the code). Naturally, traditional implementation will not reflect domain knowledge as accurately as REDK does. We show one such implementation.

Based on Table 10, an implementation class diagram which does not include the classes of Valuable Customer and Manager may be conceived. Since the perception of the
existence of these classes is only implicitly acknowledged at the organization, and has only emerged during the initial parts of requirements elicitation, a programmer may design the system using only Customer and Employee objects. Moreover, from an implementer's point of view, considering only functionality, the system may not maintain information about managers and valuable customer separately. The programmer may decide to represent managers as employees whose MaxDiscount attribute is different than zero. This design entails the implicit decision that determining if an employee is a manager can be made according to the value of the MaxDiscount variable. Similarly, a valuable customer can be determined according to the value of the EligableForDiscount variable. This design can provide for the required functionality. However, it will "hide" the domain knowledge revealed during RE. Specifically, it would be difficult to deduce from the design that managers will use the system to handle valuable customers, and regular employees will handle regular customers. It will even be hard to realize the importance of differentiating managers and valuable customers from regular employees and customers. While this information is not clearly conveyed in the implementation design, it will most likely be embedded somewhere in the information system, likely in rules implemented in the software. An example is shown in figure 21. In the figure, the logical design includes a TaskManager object. This object assigns tasks to different employees. Within the logic of this object we will find the assignment of requests according to types of handled customers.
We further use this example to illustrate how new implementation concerns emerge at the logical design phase. Those concerns further obscure the REDK. Firstly, we notice that new implementation artifacts are introduced. There is no need to get into specific implementation details, but it is reasonable that any implementation would add some artifact. Hence, for example, the employee class now may have a new procedure `UpdateCustomerQueue`, which has no meaning at the conceptual level. Moreover, new classes emerge: `RequestHandler` and `TaskManager`. The purpose of `RequestHandler` is to handle incoming calls. As well, the objects `Customer` and `Employee` incorporated new elements, which do not appear at the conceptual model. These may include the `FileHandle` and `NumOfRequestHandled` attributes of `Employee`, and the services `DisplayInfoOnScreen` and `UpdateCustomerQueue` of `Customer` and `Employee` objects, respectively.
With REDK modeled we can now examine the domain knowledge gathered during RE, and check which REDK is not present in the logical design. We also examine the logical design and check which elements are not related to REDK. Thus at this stage the differences between explicitly represented REDK and the way REDK is represented in implementation code can be analyzed.

<table>
<thead>
<tr>
<th>Difference</th>
<th>Object</th>
<th>REDK Embedded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Employee</td>
<td>The domain actually consists of managers and employees. Managers are eligible to grant discount; regular employees are not. Managers can provide extensive info. Employees provide only basic info.</td>
</tr>
<tr>
<td>2</td>
<td>Customer</td>
<td>The domain actually consists of valuable customers and regular customers. Regular customers are not eligible for any discount; valuable customers are. Regular customers are typically interested in basic info. Valuable customers need more details</td>
</tr>
<tr>
<td>3</td>
<td>Employee</td>
<td>Some properties are machine domain properties — id, FileHandle. And machine domain services – Update Customer Queue.</td>
</tr>
<tr>
<td>4</td>
<td>Customer</td>
<td>Some services are machine domain services – Display Info on Screen</td>
</tr>
<tr>
<td>5</td>
<td>TaskManager</td>
<td>Incorporates Managers service handle Valuable Customers; and Employees service Handle Regular Customers</td>
</tr>
</tbody>
</table>

If the goal-oriented approach was employed the following additional differences may be extracted.
Table 12. Differences Elicited in Goal-Based RE.

<table>
<thead>
<tr>
<th>Difference</th>
<th>Object</th>
<th>REDK Embedded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Manager</td>
<td>Managers also strive to increase valuable customers' business. Managers also try to increase company profit – their goal is to provide minimal discount while retaining valuable customers</td>
</tr>
<tr>
<td>2</td>
<td>Employee</td>
<td>Employee's major goal is to serve as many customers as possible to increase compensation</td>
</tr>
<tr>
<td>3</td>
<td>Customer</td>
<td>The domain actually consists of valuable customers and regular customers. For regular customers money well spent is fulfilled through satisfactory products. Valuable customers more important goal is to get explanation of complex features</td>
</tr>
</tbody>
</table>
6. Proof of Concept Description

6.1 General

In this chapter we describe a possible system design and a prototype implementation where the domain knowledge is represented explicitly. The design considerations discussed in this chapter are intended both to demonstrate feasibility of the ideas developed in the thesis and as well to provide designers with some guidance on how domain knowledge can be explicit and drive processing.

We briefly repeat some of the motivation for our research. We note that domain knowledge is currently intertwined with programming knowledge in information systems code. This type of implementation makes domain knowledge inaccessible to individuals lacking programming knowledge. Similarly to the way data and implementation used to be highly coupled (Ulman et al., 1991) in code before Database Management Systems (DBMS) emerged; today knowledge and implementation are highly coupled. The motivation for the work done by Codd (1982), which resulted in the relational model, was in many ways similar to some benefits that can be noticed in the proof of concept portion of our work. Codd addressed the coupling issue between systems and their databases, and suggested the relational database to draw the line between physical and logical data concerns: “the objective of providing a sharp and clear boundary between the logical and physical issues of database management…We call this the data independence objective.” (p.110). A second objective stated by Codd was “to make the model structurally simple, so that all kinds of users and programmers could have a common understanding of the data, and could therefore communicate with one another about the database. We call this the
communicability objective” (p.110 - 111). Similarly, in our proof of concept we try to decouple knowledge from implementation, and enable better communication about domain knowledge using the IS code. We claim this can support processes associated with knowledge gathering and acquisition.

In demonstrating the explicit representation of domain knowledge in IS code using the REDK meta model representation constructs of figure 10 (section 4.3.2) we take a step further and provide a way for representation in which REDK is essentially separated from other types of knowledge in the IS code. Basically, we provide a way to use REDK representation constructs, depicted in figure 10, explicitly in IS code. Enabling programmers to use these constructs and explicitly represent the domain knowledge elicited during RE processes. Many different ways to enable such use of constructs may be suggested. In our proof of concept we provide one that facilitates separation of domain knowledge in code.

To this end, a system component, termed the Knowledge Connection Engine (KCE), is defined and implemented. This component enables the explicit representation of domain knowledge. It also facilitates separation between REDK and other knowledge elements.

Within the KCE design we incorporate a Domain Knowledge Representation Layer (DKRL). Within this layer one would find the explicit representation of domain knowledge. Domain knowledge, such as the organization’s different actor types, resource types, roles, services, attributes, constraints, and goals, is represented at this layer. IS and applications such as Knowledge Management Systems of the organization interact with the DKRL. It should be noted that we do not provide a fully functional component, nor do we suggest that we account for all technical issues such a component would have in a real
industry setting. The purpose of this section is to demonstrate the possibility of implementation, issues that need consideration, and advantages that can result from such an implementation.

In our description of the KCE, the way domain knowledge is represented, and how the KCE facilitates its use, we use several figures in this chapter. The different figures focus on different perspectives of our implementation. While each figure will be discussed in detail in the remaining of this section, we now briefly review the relation between the different perspectives they focus on.

Figure 22 serves to show the role of KCE within an organization. That is, it is a high level diagram showing which systems may interact and use the KCE. After defining the general role of the KCE we will turn to explain how it is implemented to support REDK representation in code.

Figure 23 provides a high-level system overview, which shows how specific applications may use domain knowledge and how domain knowledge can derive their processing.

Figure 24 provides explanation on how specific set of classes, which we term domain base classes, are used to support knowledge representation in the KCE. Namely, domain base classes enable developers to use the REDK constructs. More specifically, the REDK constructs used in code map to the domain base classes. For example, an REDK construct such as Goal map to a corresponding Goal domain base class. These domain base classes support implementation artifacts needed for their use during system execution. For example, if we wish to store some run-time information associated with the arrival of
specific goals etc, some storage needs to be allocated. This is supported in the domain base classes.

Finally, figure 25 conceptually gives an overview of the different types of classes and components used in implementation code. These include the domain base cases, discussed above. These also include semantic classes, which refer to the domain base classes, and other classes which will be further described in this section.

![Figure 22: The Use of a Knowledge Connection Engine in the Framework](image)

**6.2 Design Principles**

To facilitate explicit representation of domain knowledge we suggest that domain knowledge should be separated from other types of knowledge in system code. We also suggest that domain knowledge should derive the processing of the information system.
Finally, we suggest that an effective way to enable domain knowledge representation is to use the ontologically-driven REDK constructs we have formalized in section 5.

We have formalized a general system design allowing us to follow these principles. We provide some details about this design in the section. We further elaborate on this issue with more detail in the next subsection.

While our general system design does not provide a complete system architecture, we occasionally use the term “architecture” to refer to this general system design.

According to this, the general system design comprises of four components (see figure 23):

- **The Conceptual Subsystem (CS):** A subsystem implementing REDK knowledge. The source code of this subsystem includes explicit representation of REDK. i.e. this subsystem consists of code defining knowledge such as the specific roles in the domain, their services, goals, etc. The knowledge incorporated in this subsystem can be used by processing subsystems, defined next. We provide more details about this subsystem in the next subsection.

- **The Processing Subsystems (PS):** A set of subsystems incorporating all other system related issues such as data type and structure information, services implementation, algorithmic details, input/output operations, device interaction, and user interface aspects. Within the source code of these subsystems no segments would embed REDK.

- **The Instance Base (IB):** An instantiation of actors, roles and resources, used during run time, of the specific application domain. The instance base allows the PS to instantiate domain related elements, such as roles, actors, and resources. The source code defining
these elements is found in the CS, and upon system execution the PS can instantiate these elements. One instantiated these instances are found at the IB.

– *The Mediator Component (MC):* A component which incorporates all knowledge associated with interaction between the other components. This component does not include domain knowledge. This component only facilitates the interaction between the PS and its IB. For example, it provides methods for creating, destroying and manipulating instantiated roles during system runtime.

The role of each component, examples of instances within components, and arrows indicating interactions between components are provided in figure 23.
In figure 24 we use UML classes to show how REDK representation constructs are supported. These classes are incorporated in the Conceptual subsystem, according to the classification above. Hence, the Conceptual subsystem incorporates classes which serve as building blocks for REDK representation. These classes are related, in a one-to-one relationship, to the REDK representation constructs of figure 10. Each construct of figure
10, has a class at the Conceptual subsystem with that name. In figure 26 we see how support for all REDK constructs is provided.

Figure 24 illustrates the use of the constructs to represent specific domain knowledge. The classes above the horizontal line reflects specific domain knowledge linked by an “is-a” relationship to REDK constructs, which are represented by classes at the lower portion of the figure. That is, the area beneath the horizontal line incorporates REDK representation constructs. The area above the line, which we called the DKRL, makes explicit use of these constructs to represent the REDK. Specifically, in figure 24, we see that a Student role in the domain made explicit at the DKRL. However, since UML classes do not naturally support the representation of REDK constructs (such as goals, stability etc), this figure only shows that the domain role student incorporates some attributes. The rest of the domain knowledge is not reflected in the figure.
To overcome this difficulty with UML, we extend on the specific REDK knowledge and its general use in the system without using UML notation. Figure 23 does not use UML notation, but rather incorporates add-hoc notation we use to illustrate REDK representation and use. While UML served well to reflect the domain based classes, in figure 23 we concentrate on other portions of the system — those that make use of the REDK
representation constructs (i.e. the DKRL), and other portions of the system, which make use of DKRL.

The diagram in figure 23 includes two core boxes, bounding the CS and the PS respectively. In the former, the domain knowledge is represented. In the latter, other elements of the system are included. The diagram also includes a third box – the mediator component, which connects the conceptual subsystem and the processing subsystem.

Within each box the diagram includes a modified UML class diagram, exemplifying some classes which may be incorporated in each of the component's code. These UML classes are not shaded. The boxes also include items depicted using shaded UML classes. These come to represent instance objects that may be created during the operation of the system. As such, these instances include examples of values that elements of the modified UML classes may include. For example, we see that the diagram incorporates a shaded UML class Actor in the instance base. Since this UML class is shaded, this UML class in effect represents an instance of and object of class Actor. As such, this instance incorporates some values of elements. Specifically the values are the different roles the actor plays – Employee and Student.

In the figure we see examples of explicitly represented domain knowledge (at the Conceptual Subsystem, in the DKRL) – such as REDK about the role student. We also see that interaction between the processing subsystem and conceptual subsystem is done using a mediating component. This component enables instantiating roles and actors, and modifying the instances, as well as handling events sent by the Conceptual Subsystem. The mediating component creates, modifies, and destroys domain objects, which are present at the Instance Base.
The Instance Base (IB) stores instantiated actors, roles and resources defined at the Conceptual Subsystem, during system run-time. This component thus enables run-time use of conceptual knowledge, such as the manipulation of the state of specific roles, actors and resources existing in the domain. These instances are managed by the Processing Subsystem, but are accessible to the Conceptual Subsystem. The Processing Subsystem maintains the Instance Base (i.e. creates, deletes and updates instances) via its Mediator Component. In figure 23, an IB with an instance of an actor playing the roles of student and employee is exemplified.

The Conceptual Subsystem is “aware” of any changes to Instance Base instances. When such changes occur, the CS reviews its knowledge about the laws, constraints, and state definitions associated with the modified instance. If it identifies an instance in an unstable state, violating a constraint, or reaching a documented state of interest, such as a goal, it will send an event, incorporating information on the type of event and associated instance, to the Mediator Component.

The Mediator Component will direct this command as a request to the appropriate implementation object. In figure 23, methods of the Mediator Component responsible for directing commands to the PS are illustrated. These are Handle Constraint Event and Handle Stability Event.

At the Processing Subsystem different implementations can be defined for the different events. For example, in case of violated constraint, an error message may be generated by the Processing Subsystem method. Alternatively, when an unstable state of a documented

---

2 Note that the system does not limit the Processing Subsystem to use roles of actors. Rather, since classes in source code often relate to roles rather than actors, direct instantiation of roles is enabled as well. Therefore, in figure 2 an instance of a student role, independent of an actor, is present.
service is reached, the implementation object will execute the methods implementing the law that has been activated.

For illustration purposes consider an inventory management system. A possible state of instability may be the system reaching an insufficient amount of inventory. In such a case some processing may be needed, including ordering new products etc. Only once the inventory reaches a predefined level again will the system reach stability.

Under such circumstances, the DKRL will incorporate a law in which the unstable state is “Insufficient inventory” and the stable state is “sufficient inventory”. Once the instance associated with recording inventory levels changes its state, the KCE will monitor whether the state of instability has been reached. If the inventory level is below the minimum level defined an event of instability will be sent to the processing subsystem. The processing subsystem will go forward with all the processes associated with balancing the inventory. As part of this processes inventory may be ordered and eventually inventory levels will change again. That is, the instance associated with recording inventory levels will changes its state. As this instance is located at the KCE the KCE will notice this change. Furthermore, if the required level of inventory is balanced again the KCE will issue a new event to the processing subsystem, notifying the processing subsystem that stability has been reached.

Note that service execution may result in additional state changes of instances in the Instance Base. As the Conceptual Component is aware of any such changes, it will also check that the outcome defined by the law (the stable state) is reached. Once the instance has reached a stable state a new event is issued to the Mediator Component, indicating successful completion of the service.
Also note that during the process of service execution, goal handling, or constraint handling, additional states of interest may be reached issuing additional events by the conceptual subsystem to the processing subsystem.

6.3 KCE Definition

Having defined the design principles and overview the operation of the system, we now formally define the KCE and its use in a general system setting, which were explained in the previous section and shown in figure 23.

In order to frame its definition in some literature based framework, we refer to literature associated with software architecture definition. As previously stated, while the KCE does not serve as a fully defined component in a full scale architecture, we looked to anchor its specification in some relevant literature.

Different definitions for the term software architecture can be found in the literature. However, the general meaning of the term is intuitively clear. A software architecture is “the structure and organization by which modern system components and subsystems interact to form systems and the properties of systems that can best be designed and analyzed at the system level” (Kruchten et al. 2006 p.23). In their seminal paper, Perry and Wolf (1992) suggested a model for defining software architectures. Namely, their model for software architectures is:

\[ \text{Software Architecture} = \{ \text{Elements, Form, Rationale} \} \]

That is, a software architecture is a set of architectural elements that have a specific form (i.e. specification of their importance), along with the rationale for the choices made in the architecture definition. Using this model, we define the context of use of our proof of concept components as follows:
**Rationale**

The rationale for the architecture used in our proof of concept is based on two core principles:

1. REDK represented in a system should be free of implementation artifacts. That is, domain models documented under the KCE should not include artifacts such as calculations, handles to devices, or network connections.

2. The content and structure of represented domain knowledge should be independent of properties of information systems using the domain knowledge. Typically, domain models are relevant to many different systems implemented within the domain. Domain models are likely to be used by different systems for different purposes. Therefore, it is important that under the proof of concept implementation the REDK of domain models will not be determined by specific systems using it. Moreover, tightly coupling domain models with specific implementations implies the incorporation of implementation concerns within the domain models.

**Elements**

Our proof of concept rationale involves the specification of its design. In our suggested implementation, a system is composed of two core subsystems: The Conceptual Subsystem and Processing Subsystem. These two subsystems are further composed of different elements as described as follows: *Conceptual Subsystem*

The Conceptual Subsystem enables the documentation of domain models and their use by interacting applications. Following its purpose, our component was designed in a
manner that enables the use of represented domain knowledge by many systems in the organization. That is, our design enables different systems to connect to one central repository of explicit domain models. As such, an organization will typically have only few Conceptual Subsystems to which different Processing Subsystems can connect. The Conceptual Subsystem is an independent subsystem, supporting interaction with different Processing Subsystems.

This independence is enabled through three elements comprising the Conceptual Subsystem: REDK, Domain Base Classes, and Routing Engine. At the core of the Conceptual Subsystem resides the REDK. The REDK incorporates the documentation of domain knowledge. I.e. the REDK includes definition of specific roles, actors, resources, etc. in the domain of discourse. The most significant property of the REDK is that it keeps an implementation free representation of domain knowledge elements. To support such a representation, the second element of the Conceptual Subsystem is required – the Domain Base Classes. The role of the domain base classes is to free the definition of the organization's domain knowledge from implementation artifacts. More specifically, each REDK meta model construct is implemented via a domain base class encapsulating all implementation elements needed to support the ability of actual instantiation of elements such as Actors, Resources, Roles, and their Attributes as they are documented in the DKRL. In other words, the domain base classes Attribute, Resource, Actor, ServiceLaw etc (figure 24). are used in the DKRL to support instantiation of domain entities without including implementation artifacts.

The final element incorporated at the Conceptual Subsystem is the Routing Engine. The Routing Engine supports event based interaction with application subsystems. Events
documented at the domain model can be handled by interacting applications through the use of the routing engine. Specifically, the routing mechanism sends event notifications to the processing subsystems once changes to unstable states and stable states are identified, goals are achieved, or constraints are violated. Notably, typically a given application would be concerned only with a subset of the documented events. It is conceivable that applications would not be concerned with goal achievements. Also, it may be that handling of unstable states would be done by different application subsystems in the organization. Therefore, not all documented events would be routed to all interacting applications. A Processing Subsystem can use the mediator component to register to a subset of events which are of interest to it events, from the Conceptual Subsystem. A specific Processing Subsystem can indicate that all events associated with a specific object are of interest. Since events are sent by the Conceptual Subsystem, essentially an implementation artifact seems to be imposed. However, the routing mechanism is implemented using Aspect Oriented Programming enabling implementation free documentation of domain knowledge. Namely, the Conceptual Subsystem will incorporate an Aspect that checks for the arrival at states of interest once attributes are changed or relations are modified.

The discussion above completes the elements of the conceptual subsystem. Next we discuss the processing subsystem.

**Processing Subsystem**

Each Processing Subsystem is composed of four elements: *Semantic Classes, Service Classes, Base Classes* and *Documentation Connection Layer*. The first three elements are associated with the assumption that the information systems using the KCE are
implemented using object-oriented methods. Under an object-oriented paradigm, typically an information system makes use of the DKRL in its semantic classes. Semantic classes are classes that are tightly related to documented domain entities. Semantic classes provide the implementation based representation of domain elements.

Within the general framework of object-oriented implementation (depicted at the left side of figure 25) the information system would incorporate interactions with the DKRL (via the KCE) in its semantic classes. Semantic classes are those classes that usually make use of the domain knowledge; however they also incorporate other programming related constructs. For example, semantic classes may be class employee or class furniture. Note however, that as opposed to DKRL elements, semantic classes will typically include implementation artifacts such as handles to devices etc. (a specific example we worked on is provided in the next subsection).

Semantic classes also relate to base classes (see figure 25). These classes refer to pure implementation artifacts such as different data structures (lists, hashtables etc) or physical operation (such as networking and outputting operations). To enable the use of the DKRL by semantic classes a connection layer between the program and the KCE is defined. Among other things, this layer incorporates procedures for connecting to the KCE, registering to events and defining procedures executed once events from the KCE are received. Basically, the connection layer encapsulates all interaction with the KCE. Figure 25 also depicts the other modules of the KCE that were described in this section. Specifically, DKRL, the use of Domain base classes, and the routing service within the KCE are indicated.

---

3 That is, while the DKRL incorporates all domain knowledge, and only domain knowledge. Semantic classes may embed some domain knowledge, and do not incorporate all domain knowledge.
A summary of the classes used in our proof of concept implementation are shown in figure 25. In this figure, adjacent areas indicate some level of interaction. The routing service is adjacent to the DRKL and Domain base classes since the routing engine accesses the DKRL to identify states that the processing subsystem should be notified on. It also relates to the base classes which incorporate the aspect suggesting when evaluation of states should take place. The separation of KCE from the processing classes is illustrated by having the KCE Connection layer as the only point of contact between the two. The shaded Boundary area illustrates the idea that Base classes have no relation with any element of KCE.

<table>
<thead>
<tr>
<th>Semantic Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee: (uses DKRL definition of Role Employee)</td>
</tr>
<tr>
<td>Meeting room: (Uses DKRL definition of Service Meeting room)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KCE Connection Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain Layer-KCE</td>
</tr>
<tr>
<td>DKRL (Meeting room: resource, Manager: role, Employee: role)</td>
</tr>
<tr>
<td>Domain Base Classes (Actor, Resource, Goal, Attribute etc.)</td>
</tr>
<tr>
<td>Routing Service (AOP)</td>
</tr>
</tbody>
</table>

**Figure 25: System classes and components**

---

4 Note that the figure only shows classes and not instances. Therefore the instance base is not depicted.
The *Form* aspect of an architecture describes the relative importance of elements in the architecture. The form of an architecture also relates to the importance of relationships between elements of the architecture. Since in our implementation we do not provide a full scale architecture, but rather describe the essential elements of operation, all elements described in this document are necessary for the workings of our implementation.

### 6.4 Knowledge Connection Engine (KCE) Summary

The KCE supports three basic functions:

- Representation of domain knowledge without implementation considerations.
- Access and use of domain knowledge by any application.
- Interaction with applications by an event notification mechanism.

The KCE can be used by all information systems. As such, an organization will typically have only one KCE and different applications can make use of the same KCE. To support such use of the KCE, the proof of concept has been implemented to be an independent process interacting with different applications over network connections.

At the core of the KCE resides the Domain Knowledge Representation Layer (DKRL). The DKRL incorporates all domain knowledge elements as defined in the previous section. Updates and queries on domain knowledge are materialized by accessing this layer. The most significant property of the DKRL is that it keeps an implementation free representation of domain knowledge.
The KCE also includes an event routing mechanism. This mechanism enables the notification of interacting applications about the occurrence of events at the domain level. Specifically, the routing mechanism generates and routes events once an object reaches state of interest, such as the arrival on an unstable state, the accomplishment of a goal, or the violation of constraint. The design does not impose constant processing of events. Rather, each processing component needs to specify which events are of interest to that component. In other words, not all events in the system would be routed to all interacting applications. In order to be notified of events, interacting applications register to the routing mechanism and indicate the events of interest at the implementation level.

Finally, the use of the KCE by an application requires support from the processing components as well. In our research, we assume the information systems are implemented using object-oriented constructs.

6.5 Implementation

To show feasibility of use of the KCE, we have modified an implemented case study adopted from the literature (Barker and Palmer 2004). The case study was altered so it will make use of the DKRL, represented at the KCE.

6.5.1 Knowledge Connection Engine Interface

In addition to the details provided in the previous section, we have augmented the KCE with functionality that enables its use as a "domain events control tool". That is, we incorporated within the KCE functionality which enables real-time reporting about states of interest in the domain. Specifically, regardless of any connected application, the KCE
reports about domain entities achieving goals, violating laws, arriving at unstable states, and returning to stable states. Moreover, whenever a state of interest is arrived at, additional information regarding the specific laws\goal documented this state of interest is provided.

On top of reporting domain related information. The KCE also reports technical information about its operation. The KCE monitors whether the Events engine is functioning properly or if event notification problems occur. The KCE also monitors whether connected applications have abnormally been disconnected and thus whether related resources can be freed and domain events still need to be routed to the inspected application.

6.5.2 Application Subsystem Modification

In order to examine the interaction of an application with the KCE, we have adopted a sample system from the literature. Specifically, we used the student registration case provided by Barker and Palmer (2004). We changed this case so that our modified version of the system would make use of the KCE.

In general, Barker and Palmer (2004) provide a system that handles tasks associated with student enrolment services. Through its user interface students can enroll to, and drop out of different sections. This application incorporates student, professor, and person classes. These classes have been modified to work with the KCE and were represented as roles. In the explanation of our modification we concentrate on student class for the purpose of demonstrating the advantages of explicit representation of REDK.

Simplifying syntactic elements, the code of the original application defined the student class as incorporating the attributes and methods shown in figure 26:
<table>
<thead>
<tr>
<th>Code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>String major;</td>
</tr>
<tr>
<td>String degree;</td>
</tr>
<tr>
<td>Transcript transcript;</td>
</tr>
<tr>
<td>ArrayList attends;</td>
</tr>
<tr>
<td>String password;</td>
</tr>
<tr>
<td>Void Display()</td>
</tr>
<tr>
<td>String ToString()</td>
</tr>
<tr>
<td>Void DisplayCourseSchedule();</td>
</tr>
<tr>
<td>Bool IsEnrolledIn(Section s)</td>
</tr>
<tr>
<td>Void DropSection(Section s)</td>
</tr>
<tr>
<td>Void AddSection(Section s)</td>
</tr>
<tr>
<td>Void PrintTranscript()</td>
</tr>
<tr>
<td>Bool IsCurrentlyEnrolledInSimilar(Section s1)</td>
</tr>
<tr>
<td>IEnumerator GetEnrolledSections()</td>
</tr>
<tr>
<td>Int GetCourseTotal()</td>
</tr>
<tr>
<td>Void ParseData(string line)</td>
</tr>
<tr>
<td>Void ParseData2(string line)</td>
</tr>
<tr>
<td>Bool StudentSuccessfullyInitialized()</td>
</tr>
<tr>
<td>Bool Persist()</td>
</tr>
<tr>
<td>Public bool ValidatePassword(string pw)</td>
</tr>
<tr>
<td>Public ArrayList GetSectionsEnrolled()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Some Explanation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two string attributes defined in the class</td>
</tr>
<tr>
<td>A transcript object is part of the class</td>
</tr>
<tr>
<td>Attends property is of type ArrayList</td>
</tr>
<tr>
<td>Password is an attribute of type String</td>
</tr>
<tr>
<td>Display is a method returning void</td>
</tr>
<tr>
<td>ToString is a method returning void</td>
</tr>
<tr>
<td>DisplayCourseSchedule is a method returning void;</td>
</tr>
<tr>
<td>IsEnrolledIn is a method returning bool and accepts as input a section object</td>
</tr>
<tr>
<td>DropSection is a method returning void and accepts as input a section object</td>
</tr>
<tr>
<td>AddSection is a method returning void and accepts as input a section object</td>
</tr>
<tr>
<td>PrintTranscript is a method returning void;</td>
</tr>
<tr>
<td>IsCurrentlyEnrolledInSimilar is a method returning bool and accepts as input a section</td>
</tr>
<tr>
<td>GetEnrolledSections is a method returning IEnumerator is a method returning int</td>
</tr>
<tr>
<td>GetCourseTotal is a method returning int</td>
</tr>
<tr>
<td>ParseData is a method returning void and accepting a string</td>
</tr>
<tr>
<td>ParseData2 is a method returning void and accepting a string</td>
</tr>
<tr>
<td>StudentSuccessfullyInitialized is a method returning bool</td>
</tr>
<tr>
<td>Persist is a method returning bool</td>
</tr>
<tr>
<td>ValidatePassword is a method returning bool and accepts a string</td>
</tr>
</tbody>
</table>

Figure 26: Student class definition of original application (adapted from Barker and Palmer 2004)

We have intentionally grouped only segments of code defining attributes and services. We have omitted segments associated with method implementation. We have done so to enable some basis for comparison between the DKRL based design and the original design. Hence, figure 26 stresses elements of code which can potentially be understood also by

161
individuals with some minimal programming knowledge. This is done to examine system
code as a documentation artifact, facilitating communication between all stakeholders.

Note that even if all stakeholders had excellent programming knowledge, some
deficiencies in viewing source code as a source of domain knowledge would still be
evident. For example, methods intimately related to the operation of the system are not
distinguished from methods associated with the definition of domain entities such as
students. More specifically, methods such as ParseData, ValidatePassword, Display,
StudentSuccessfullyInitialized, and ToString, do not have a meaning outside the discussed
application. However, methods such as AddSection, DropSection, and GetSectionsEnrolled
have a meaning in the application domain; and thus not only to programmers, but also to
different stakeholders. This mixture between application related methods and domain
related methods in the original application is a potential source for communication
problems between the different stakeholders of the system. Arguably, it may be that with
prior domain knowledge and effective method naming conventions, the domain of different
methods could sometimes be distinguished. However, one of the important advantages of
using explicit REDK is in supporting obtaining of domain knowledge rather than demand
domain knowledge as a prerequisite for understanding the meaning of code. Contrary to an
REDK based design, when using the source code of the original application to acquire
domain knowledge, stakeholders in the organization may mistakenly assume that students
in the domain actually parse data.

Figure 26 serves to demonstrate some more deficiencies when using source code as a
source for domain understanding under current programming standards. For example, often
even domain related methods may be impossible to understand. While the meaning and use
of methods such as AddSection or DropSection can be generally understood, some methods may remain ambiguous when lacking programming knowledge. In this case, IsCurrentlyEnrolledInSimilar has a domain meaning— the method checks if a student is enrolled in a section of a particular course. However, without programming knowledge it is impossible to know when and how it is used.

After we have modified the application, to use the KCE, the student class was decomposed into three classes – ImplementationStudent, StudentInteractor, and DKRLStudent. This decomposition is depicted in figure 27. ImplementationStudent incorporates the elements of the original student class which relate solely to implementation. DKRLStudent incorporates all documentation associated with students in the domain and is represented in the documentation subsystem.

DKRLStudent demonstrates the most important characteristic of the new design - its ability to document domain knowledge explicitly and separately. DKRLStudent contains no methods but rather defines the student role, incorporating states, laws, and attributes. While we have not designed a new programming grammar, as illustrated in figure 27, since the documentation does not deal with method implementation, a natural language can be applied for documentation.

Also seen in figure 27, is that StudentInteractor serves as a bridging element between the implementation frame of reference and the domain frame of reference. The StudentInteractor incorporates all the methods which interact with the explicitly documented knowledge. With respect to that, it may add implementation related artifacts to the documentation. For example, it can be seen that GetEnrolledSections adds an implementation related data type, IEnumerate. IEnumerate enables efficient use of the
documentation, but is not domain related and therefore is addressed only at the interacting object.
Class Implementation Student (In Processing Subsystem)
/*Class with Implementation elements of Students and access to domain elements */
String password;
//Handle to Mediator Class (which interacts with KCE)
Documentation Student Interactor Interactor
Void Display()
String ToString()
Void DisplayCourseSchedule();
void PrintTranscript()
int GetCourseTotal()
Void ParseData(string line)
void ParseData2(string line)
bool StudentSuccessfullyInitialized()
bool Persist()
public bool ValidatePassword(string pw)
public void UpdateRegisteredSections();

Class Student Interactor (Mediating Class)
Documentation Student Documentation
bool IsEnrolledIn(Section s)
void DropSection(Section s)
void AddSection(Section s)
public ArrayList GetSectionsEnrolled()
Public HandleEvent (EventDetail E)
IEnumeratorator GetEnrolledSections()

Some Explanation: These methods provide the implementation of services documented in the CS
Password is an attribute of type string
Interactor is an object enabling interaction with domain knowledge about students
Display is a method returning void
ToString is a method returning void
DisplayCourseSchedule is a method returning void;
PrintTranscript is a method returning void;
GetEnrolledSections is a method returning
ParseData is a method returning void and accepting a string
ParseData2 is a method returning void and accepting a string
StudentSuccessfullyInitialized is a method returning bool
Persist is a method returning bool
ValidatePassword is a method returning bool and accepts a string
UpdateRegisteredSections is a method responsible for processing the service RegisterToOne service

Some Explanation:
Methods defined in the interactor need to interact with the KCE to modify domain related data, or arrange domain related data in special structures
Documentation is a handle to the Domain Student
IsEnrolledIn is a method returning bool and accepts as input a section object
DropSection is a method returning void and accepts as input a section object
AddSection is a method returning void and accepts as input a section object
GetEnrolledSections is a method returning

Figure 27a: New implementation of student related classes (Mediating, and Processing)
DKRLStudent (incorporating REDK about the student role, in DKRL)
Implementation of services is done in the processing class demonstrated in figure 29a
Student is a role with:
- Has Property major;
- Has Property degree;
- Has MutualProperty transcript NotModify
- Has ManyRelationship with section, named "attends"
- Has a State:
  - RegisteredInOneSection
    - if(many relation "attends". Size = 1)
      - State is unstable
    - Else
      - State is stable
- Has a State:
  - MissingCourseForGraduation
    - if (Credits == FullCredits-1 )
      - State is Unstable
    - Else
      - State is stable
- Has a State:
  - RegisteredInSectionMoreThanOnce
    - for each Section participating in the "attends" relation:
      - if the number of times the section participates in "attends" >1
        - State is unstable
    - State is stable
- Has a Service RegisterToOne
  - UnstableState = RegisteredInSectionMoreThanOnce
  - StableStable = not RegisteredInSectionMoreThanOnce
- Has a Service RegisterToOneSection
  - UnstableState = MissingCourseForGraduation
  - StableStable = RegisteredInOneSection

Figure 27b: New implementation of student related classes (Conceptual)
The explicit representation is also able to capture much of the domain knowledge previously embedded at the application and not accessible to non-programmers. The domain elements, previously embedded in \textit{IsCurrentlyEnrolledInSimilar}, are now made explicit in the definition of the \textit{RegisterToOne} law. The previously embedded domain elements are now documented by defining the state \textit{RegisteredInSectionMoreThanOnce}. From the law it is also clear how the elements are used and when: if the state \textit{RegisteredInSectionMoreThanOnce} is reached by a student, the application subsystem will be notified to take an action to bring the student to a state in which it is registered only once in the section. This is done in the \texttt{UpdateRegisteredSectionsMethod}. It is also interesting to note the flexibility in changing the DKRL. For example it can easily be stated that the stable state in the discussed law would be \textit{RegisteredInOneSection}. Now the application subsystem will have to handle the event of \textit{RegisteredInSectionMoreThanOnce} differently in order to arrive at a stable state.

\textbf{6.5.2.1 Lessons Learned from Proof of Concept}

The proof of concept was implemented to examine a practical way for explicitly represented REDK domain knowledge. While accomplishing this objective a few lessons regarding the challenges in explicit REDK representation have been learned.

First, we found that our proof of concept requires a fundamentally different way of approaching implementation design. That is, when using the prevailing object-oriented paradigm, procedural programming is naturally used. However, when aiming to explicitly represent domain knowledge, a blend of declarative statements needs to be supported. This became very apparent when redesigning the case where processes needed to be altered in

\footnote{5 Since according to our ontological analysis, laws manifest services we refer to service definition as laws.}
order to support explicit representation of domain knowledge. Specifically, unstable and stable states, which could not be inferred from program code needed to be assumed in our case.

To demonstrate, in the original program, the registration process consisted of one method, executed upon request from the application user. In the modified case, rather than have the process avoid unstable states, we created a set of services executed upon the occurrence of unstable states. For example, one such unstable state is arrived at when a student is registered in two sections that take place at the same time, or when a student is registered to two sections of the same course.

Related to this, it appears that current object-oriented programming practice fails to use the Actor and Role concepts effectively. That is, it is common practice that in source code there is no entity that acquires and loses roles dynamically. Rather each actor is defined by one role it possesses throughout its lifetime. For this reason we have enabled the direct instantiation of roles in the instance base. Instantiation of roles implies that the processing subsystem assumes that actors to not change acquire, lose, or play multiple roles simultaneously.

Another issue that came about is that much of the domain knowledge cannot be extracted from implementation design. For example, during the modification of the case it was challenging to define if certain states are states of instability or constraints. That is, a state such as “registered to two sections taking place at the same time” may be a thought of as a violated constraint or as a state of instability requiring a service to take place. Only during the Requirements engineering process is it made clear if this is a constraint that will
never happen in the system, or if this state may be arrived at, but will require some processing till a complementary stable state is arrived at.

As well, while we were able to document the REDK in our proof of concept, we did not completely free the processing subsystem from implicit incorporation of some of the domain knowledge within it. For example, while in the conceptual system there is explicit definition of states of stability that should be arrived at once an unstable state is reached, the arrival at these stable states is facilitated by the processing subsystem. Without embedding the appropriate domain knowledge within the processing subsystem a situation in which the stable state is not arrived at, can come about. Note however, that under the "architecture" and tools developed for the proof of concept, such situations are monitored and noticed by the system, thus providing indication of some possible inconsistencies in the system implementation. Therefore, an additional advantage of our design is in its ability to check if processing elements properly reflect domain laws.
7. Summary

Domain knowledge is a concept commonly applied and referred to in the Information Systems development literature. However, a clear understanding of the content of domain knowledge has not yet been established. The term “Domain knowledge” has different meanings in different fields of research. When narrowed to the field of IS, domain knowledge has been described as “an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area” (Jacobson et al. 1999). This is most often considered as the application area for which we develop software systems (Ruben 1990). This makes domain knowledge highly related to RE. Specifically, obtaining proper domain knowledge is recognized in software development as a critical factor for accomplishing the goal of RE, namely identifying complete, consistent, and accurate needs (Sutcliffe and Maiden 1998). In other words, during the requirements engineering process domain knowledge must be acquired. In this process, knowledge about the purpose of the information system and about the domain of discourse is established using different types of models. This knowledge shapes the implementation of the information system, and is eventually manifested in software code.

The purpose of this research was to advance our understanding of the meaning of the term domain knowledge, to define its constructs, and to enable its representation based on ontological guidelines. Such an understanding can help us relate the processes taking place during Requirements Engineering to the actual system code. As well, it can enable the use of domain knowledge gathered during RE in subsequent phases of the system lifecycle.
In this research we have examined the term domain knowledge in different phases of the system development processes. Given the objective of RE, it is the process most intimately related to the accumulation of domain knowledge. We have therefore conducted an analysis of domain knowledge discussion and use within the RE literature.

We have found that different requirements engineering approaches emphasize the modeling of different aspects associated with both the domain and the system. Four general categories of modeling approaches used in RE are Data Modeling, Behavioral Modeling, Enterprise Modeling, and Domain Modeling. These approaches are not mutually exclusive. Rather, each approach takes a different view as to the aspect that should be emphasized to facilitate requirements elicitation.

Data modeling approaches deal with understanding the information that needs to be stored, maintained, and managed by the information system. Behavioral modeling approaches deal with the dynamic or functional behavior of stakeholders and systems, both existing and required. Enterprise modeling approaches deal with understanding an organization's structure, the business rules that affect its operation, and the goals, tasks and responsibilities of its members. Finally, Domain modeling approaches deal understanding the impact of the system on the domain, and the way it helps control the domain.

We therefore analyzed the constructs used in these different approaches and identified those elements of requirements engineering knowledge that relate to the domain.

This analysis results in a set of domain-related RE constructs. We identified the relationship between constructs used in different disciplines within the field of RE. We found where constructs overlap and how they relate to one another. Based on this, we were
able to provide a meta model unifying the different RE constructs used in different RE approaches.

We then set out to find ways to effectively facilitate representation of DK with these constructs so that they can be used in subsequent phases of the development process.

As knowledge is recognized as an increasingly important asset in today's competitive world, facilitating source code as a new repository of domain knowledge can have potential benefits to many organizations. Specifically, within each organization possessing knowledge assets, explicit representation of accumulated domain knowledge has potential to contribute to the different knowledge seeking stockholders. These can range from the system analyst, to the development programmer, to the maintenance programmer, to the customer support representative, the management, and more. In our work (Chapters 4, 5, and 6), we show how developers may contribute to domain knowledge documentation.

Through an RE case (Chapter 4) we demonstrated how system analysts may help in documentation of domain knowledge. With a more detailed case (Chapter 5) we further demonstrated representation of DK using the constructs of the meta-model which integrates concepts from the different RE approaches. In this chapter we show how REDK documentation can help analyze the correspondence between actual IS design and the documented domain knowledge. In Chapter 6, we demonstrated how programmers may document REDK in system code and what components can support them. In this chapter we also demonstrated how programmers may be able to understand domain knowledge by examining knowledge-driven coded systems. To facilitate this use of system code, we proposed a system design in which explicit domain knowledge representation is enabled. The represented domain knowledge is based on our ontologically-driven representation
principles, which lead to an IS design where represented knowledge drives processing during system runtime.

We believe that the novelty of this work is in examining information systems in a unique way - as potential sources of domain knowledge. Our software model creates a new source of knowledge that can be tapped into.

It should be noted that while we suggest many benefits associated with explicit domain knowledge representation there of course may be costs associated with its representation. In ontology driven IS developers will need to be highly skilled in understanding the domain and effectively separating domain knowledge from implementation knowledge. Such implementation may also have costs in terms of time to completion. Finally, the creation of IS in which domain knowledge is explicitly represented should be carefully evaluated against other possibilities, such as using the work force to provide more functionality features rather than carefully model the domain. Indeed, all these issues are some promising venues to consider in future research.

This research establishes basis for potential use of knowledge which the organization may have traditionally considered tacit. As stated by Zack (1999): “Knowledge may be inherently tacit or seem tacit because no one has yet articulated it…” “Organizations often do not challenge the way they store, treat or pass on knowledge, which may result in managers blindly accepting the apparent tacitness of some types of knowledge”.

Within the scope of this thesis we have covered a wide range of aspects associated with domain knowledge. We began with meta-modeling RE domain knowledge constructs. We provided a definition of the domain knowledge constructs in the context of requirements engineering. We continued with ontology-driven representation of domain knowledge
constructs and demonstrating the use of the constructs to explicitly model requirements engineering domain knowledge. We further suggested how design procedures, in which domain knowledge is explicitly represented, can be framed within the model driven architectures framework. Finally, we demonstrated a possible way for incorporating explicit representation of domain knowledge in final implementation design, such that documented DK drives processing.

For the future, we believe this study opens the way for more research on the system development process as a domain-knowledge based activity. By better understanding the domain knowledge constructs, common grounds for discussion and ongoing research can be established. By addressing the fact that requirements engineering domain knowledge can be explicitly represented in the system code, information systems can now be examined not only according to their supported functionality, but also according to their potential contribution as sources for knowledge management in the organization.

Additionally, the way documented knowledge can be used to facilitate development and information gathering by knowledge workers can be researched. In our work we showed how analysts and programmers may benefit from REDK documentation, and use REDK representation constructs to document domain knowledge. In future research, it is possible to explore IS development processes that facilitate the creation of Information Systems conforming to our proposed system design principles. More specifically, as our literature review suggests, an implicit assumption of most current practices is that technical considerations increasingly shape the evolving design. The shift of focus to implementation details leads to a gradual loss of the explicit conceptual knowledge. This knowledge, well reflected in earlier stages of development, is progressively hidden in programming details.
as development advances. In the end, with common approaches, it is not represented explicitly in the final implementation. Thus, we believe a different design approach is needed.

Specifically, we suggest that once knowledge is formulated during RE, it can be used throughout the development process, to compare and evaluate different design documents. In the finale stages of development, this knowledge can be represented in the implementation code.

Several ways to pursue research in this direction can be suggested. As indicated in our literature review, one possible software development framework, which can help such work, is Model Driven Architecture (MDA). The MDA development life cycle is similar to the traditional life cycle (described in figure 29). MDA is intended to take part in all stages of the development process to enable a structured transformation between models (from requirements to code). However, MDA has been dominantly applied to transformations between the logical design and implementation code. This probably originates from difficulties in providing transformation rules between conceptual models and logical designs. The gap between the two is too wide. We create conceptual models to understand and communicate about the domain of discourse. On the other hand, logical designs incorporate a large amount of implementation artifacts unrelated to the original domain. However, preserving the original domain knowledge during the development transformation may facilitate the transformation between the conceptual and logical designs (see figure 30). In such a framework there might be no need for pre-set mapping rules between the conceptual and logical models. Under such research, the mapping rules
could possibly be created dynamically and would differ for each specific system. These ideas are depicted in figure 30.

![Figure 29: Conventional Development Process and MDA Concepts](image)

![Figure 30: Framework Development Process](image)

Another venue for future research is in using system code as a source for gathering tacit knowledge. While this research primarily discusses explicit domain knowledge gathered in requirements engineering, examining IS code as a source for tacit knowledge can be a natural extension of this work. Specifically, during the SA&D process system analysts obtain also what is considered to be tacit knowledge in the organization. The potential
tacitness of some knowledge obtained is evident when considering the methods employed to acquire it. Iterations of interviews and observations are some of the methods employed by the system analysts during the analysis phase. As suggested in the previous sections, large quantities of this knowledge are eventually encoded in software.

Another possible topic for future research is to examine different ways to explore the explicit domain knowledge documented. A possible application that falls under this scope may be to develop a query language designed to enable DKRL based knowledge exploration. An additional study may incorporate the use of DKRL to derive specifications of processing components. Along these lines, many additional future studies may be suggested.

Finally, there are some additional advantages to our approach. By explicitly representing domain knowledge and allowing all systems to interact with the domain knowledge we establish a common vocabulary for all applications. Additionally, the common use of domain knowledge enables central control for the validation of correct operation of processing components. It also may serve as a central area of integration between different systems.

In summary, we believe that this dissertation advances our understanding about RE processes, their use of domain knowledge, and the way domain knowledge comes to play in subsequent development stages. The dissertation develops some practical ways for facilitating the use of requirements engineering domain knowledge, and demonstrates the applicability of the developed ideas. We hope this understanding can open new directions for research and possible improvements in knowledge processes at the deploying organizations.
Reference


Bernus, P., Some Thoughts on Enterprise Modelling, Production Planning & Control, 12, 2001, 110-118.

Blum, B.I., Beyond Programming, to a New Era of Design, Oxford University Press, NY, 1999


Bratko, I., Muggleton, I.S., Applications of Inductive Logic Programming, Communications of the ACM, 38, 1995


180


Gamma, E., Helm, R., Johnson, R., Vlissides, J. M., Design Patterns- Elements of Reusable Object-Oriented Software, Addison-Wesley, Boston, USA, 1995


Guarino, N., Concepts, Attributes and Arbitrary Relations, Data & Knowledge Engineering, 8, 1992, 249-261


Jackson, M., Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices. Addison Wesley, 1995


Jackson, M., Zave, P. Domain Descriptions. 1st International Symposium on Requirements Engineering (RE'93), San Diego, USA, 1993, 56-64.


Kaindln, H., Difficulties in the Transition from OO Analysis to Design. IEEE Software. 16:5, 1999, 94-102


van Lamsweerde, A., Willemet, L., Inferring Declarative Requirements Specification from Operational Scenarios, IEEE transactions on Software engineering, 24:12, 2000, 1089-1114


Paetch, F., Eberlin, A., Maurer, F., Requirements Engineering and Agile Software Development., Proceedings of the twelfth IEEE international workshops on enabling technologies, 2003, 308-313


Reeves, W. J.: What is Software Design?. C++ Journal, 2:2, 1992


Ruben, P. D., Domain Analysis an Introduction, ACM SIGSOFT Software Engineering Notes, 15:2, 1990, 47-54


Shlaer, S., Mellor, S., Object-Oriented Systems Analysis: Modeling the World in Data. Prentice Hall, 1988


Sommerville, I., Software Engineering. Addison-Wesley, Wokingham, 1989


Steimann, F., On the Representation of Roles in Object-Oriented and Conceptual Modelling, Data and Knowledge Engineering, 2000

Tarski, A., On the Concept of Logical Consequence. In Logic, Semantics, Metamathematics, Oxford University Press, 1956

Thompson, C,: Living an Enterprise Model. Database Programming and Design , 1993

Tru, H.C., Annotated Fuzzy Logic Programs, Fuzzy Sets and Systems, 2000


