Toward a Query Language for Organizational Processes

by

Henry Tang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Electrical Engineering and Bachelor of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1998

© Henry Tang, MCMXCVIII. All rights reserved.

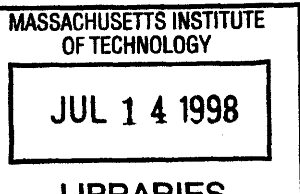
The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author ....
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by ....
Thomas W. Malone
Patrick J. McGovern Professor of Information Systems
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Eng



Toward a Query Language for Organizational Processes

by

Henry Tang

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 1998, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Electrical Engineering and Bachelor of Science in Computer Science

Abstract

This thesis discusses the development of a Process Query Language (PQL) for the Process Handbook. The Process Handbook, developed at the MIT Center for Coordination Center, is practical realization of the concepts in process modeling to help people invent, redesign, and learn about organizational processes. The PQL project involves developing a intuitive language for specifying complex process queries. PQL is intended to be (1) high-level, (2) precise, (3) extensible, (4) efficient, and (5) database independent. This thesis focuses on the design, syntax, and usage of PQL.

Thesis Supervisor: Thomas W. Malone
Title: Patrick J. McGovern Professor of Information Systems

Acknowledgments

I would like to thank many people for their support and guidance throughout my graduate study. First, I would like to thank my thesis supervisor, Professor Thomas Malone, for giving me the opportunity to work with a group of really wonderful people at CCS and for his flexibility in allowing me to pursue a thesis project that I am interested in. My deepest thanks go to my mentors John Quimby and Mark Klein for their immeasurable help and guidance throughout my project. The other members of CCS have been tremendously supportive and helpful as well. Thanks to George Herman, Avi Bernstein, and Zia Ahmed who have made working in CCS and my thesis fun and stimulating. I also want to thank the people on the "help" instance for helping me with the format of this thesis.

Work and research do not just stop at the door of the center and I would like to thank the many friends of mine without whose support and love, I would not have been this successful in academics, career, my research and thesis. First I would like to thank Calvin Yuen and Vuong Nguyen who first got me interested and subsequently led me to work at CCS. Special thanks also go to Chris Leung, my long-term friend, who has helped me in many ways, Karen Chang, who has been like a sister to me and has given me so much support, love and inspiration to succeed, Marlon Abayan at Salomon Fixed Income Arbitrage, for his guidance in my career search process, and Ting Luo, who has been my source of support and guidance recently. There are many people who have made an impact in my life that I would have to leave out over here due to space limitation. But be assured that your names reside in my heart and I love you all.

Last but most importantly, I want to thank my parents, who had given up their professions and traveled half way around the globe so that I can be here, have written this thesis, and have an exciting career after graduation. I dedicate my work, past, present, and future, to them.

Contents

1 Introduction **10**

1.1 Motivations for PQL 10

1.1.1 Process Redesign/Editorial Purposes 11

1.1.2 Dynamic Folders/Sharable Resources 11

1.1.3 High Level Query Interfaces 12

1.2 Queries we would like to make 12

1.2.1 The Internet Consultant's Query 13

1.2.2 The Operations Manager's Query 13

1.2.3 The Human Resources's Query 13

1.3 Organization of This Thesis 14

2 Background: The Modeling of Organizational Processes **17**

2.1 Decomposition and Specialization 17

2.1.1 Decomposition 18

2.1.2 Specialization 18

2.2 Dependencies and Coordination 22

2.2.1 Dependencies 23

2.2.2 Ports and Connectors 24

2.2.3 Managing Activities 27

3 Current Methods for Querying Process Handbook **29**

3.1 Find by Navigation 29

3.2 Problems with Find by Navigation 31

3.3 Search by Keyword 32
3.4 Problems with Keyword Search 32
3.5 Query by SQL 35
3.6 Problems with Query by SQL 36

4 Toward a Process Query Language (PQL) 38

4.1 The Design of PQL 38
4.1.1 Design Goals and Issues 38
4.1.2 PQL Architecture 40
4.1.3 The Two-pass Design 41
4.1.4 An Entity-Relation Approach 44
4.1.5 Matching ER Graphs 46
4.2 PQL Syntax and Simple Queries 48
4.2.1 Entity Operation 49
4.2.2 Attribute Operation 50
4.2.3 Relation Operation 51
4.2.4 Select Operation 54
4.2.5 Logical Operation 54
4.3 Complex Queries 56
4.3.1 The Internet Consultant's Query 57
4.3.2 Operations Manager's Query 61
4.3.3 Human Resources's Query 61

5 Evaluation 63

5.1 High Level 63
5.2 Database Independent 64
5.3 Increased Precision 64
5.4 Potential UI Capability 64
5.5 Optimizable 65

6 Future Work 66

6.1 Implementation of High Level Interfaces 66
6.1.1 Graphical Process Query Language (GPQL) 67
6.1.2 Natural Language Query (NLQ) 67
6.2 Language Extensions 67
6.2.1 Expressing Dependencies in Queries 68
6.2.2 Qualitative and Quantitative Queries 68
6.3 Optimizations 69
6.3.1 Query Re-ordering 69
6.3.2 Heuristic Search 70
6.3.3 Inexact Matches 70

A The Process Handbook Architecture **72**
A.1 Database Schema 72
A.2 Object API Middle Ware 74
A.2.1 The Purpose of the Object API 75
A.2.2 The Objects of the API 76
A.2.3 Extensions to the Object API 78
A.3 Beyond Object API 79
A.3.1 PH_WEB 80
A.3.2 PH_WIN 81

B Search Algorithms **84**
B.1 Breadth-first Search 84
B.2 Depth-first Search 85

C PQL Implementation Notes **87**
C.1 Overall Design 87
C.2 PQL Objects 89
C.2.1 PDB 89
C.2.2 Pair 89
C.2.3 BSet 90

C.2.4 BList 92
C.2.5 QList 93
C.3 PQL Functions 94
C.3.1 Parser 94
C.3.2 List 96
C.3.3 Search 97

List of Figures

2-1 The Process Compass (Adapted from Process Handbook Web UI) . . 18

2-2 The decomposition of Sell Product (Adapted from [6]) 19

2-3 The representation of a generic sales process (*Sale Product*) and its specializations (*Sell by Mail Order* and *Sell in Retail Store*). (Adapted from [6]) 20

2-4 Specializations of the activity Sell Something grouped by bundles [Sell how?] and [Sell what?]. (Adapted from [6]) 21

2-5 A tradeoff matrix showing rough estimate of advantages and disadvantages of different specializations for the generic sales process. (Adapted from [6]) 22

2-6 Decomposition and dependency (Adapted from [4]) 23

2-7 Decomposition of *Make Coffee* with dependencies expressed. (Adapted from [4]) 25

2-8 The semantics of multiple levels of decomposition, vertical connectors and dependency with multiple producers and a single consumer. (Adapted from [4]) 26

2-9 The changes to the decomposition of A after a managing activity is set for the dependency Depe 1. (Adapted from [4]) 28

3-1 Process specialization hierarchy from the Web Process Handbook. . . 30

3-2 Search interface from Web Process Handbook 33

3-3 Search interface from the Windows application of the Process Handbook 34

4-1 PQL/PH Architecture 41

4-2 Two-pass design of PQL 42
4-3 An Entity-Relation Diagram 46
4-4 PQL Search Tree 47
4-5 Example of PQL Entity Query 49
4-6 PQL Result Format 50
4-7 Example of PQL Attribute Query 51
4-8 Example of PQL Decomposition Query 52
4-9 Example of PQL Specialization Query 53
4-10 Example of PQL Query with SELECT...FROM 55
4-11 Example of PQL Query with Logical Operator 55
4-12 Query: "What are the different ways of selling something that involves the Internet?" 57
4-13 Query: "What are the different ways to identify potential customers if you want to sell over the Internet?" 58
4-14 Specialization of *Identify Potential Customer* 59
4-15 Decomposition of *Sell Over Internet* 59
4-16 Decomposition of *Sell Over Internet Mall* 60
4-17 Query: "What are all the sales processes that involve dealing with DARPA?" 61
4-18 Query: "What are all the ways of identifying sources that is used by a type of hire human resources?" 62

A-1 The tables of the Process Handbook Database Schema 73
A-2 The three-tier process handbook architecture. (Adapted from [9]) . . 75
A-3 Properties and methods for API objects (Adapted from [4]) 77
A-4 Components that are built on top of the Object API middleware. (Adapted from [9]) 80
A-5 A snapshot of the Web Process Handbook interface. 81
A-6 PH_WIN Viewers and Editors. 83

C-1 PQL Implementation Modules 88

Chapter 1

Introduction

The goal of this thesis is to develop a Process Query Language (PQL) for the Center for Coordination Science (CCS) Process Handbook.[6] It aims to provide a more intuitive language tailored to describing powerful process queries using the Process Handbook model. The goal of PQL is to provide a better way to query processes than the current method of using Structured Query Language (SQL) to query the Process Handbook at the implementation level. There are many advantages to having PQL. First, PQL interacts with processes at the correct abstraction level (process objects rather than the underlying relational database representation). So it is more intuitive for describing process queries. Second, PQL provides ways to specify queries with very complex process relationships that were previously not practical. These advantages and many others will be discussed in detail in later sections. One important purpose that PQL serves is to separate the process objects from its underlying representation. PQL will establish a standard way of querying processes that allows for future changes to the process representation and allows end user software to be more modular.

1.1 Motivations for PQL

The challenge is to design a highly focused, easy-to-define query language for the retrieval of the information we need from a large repository of complex conceptual process knowledge. The realization of PQL will set the ground for many powerful

new ways of using the Process Handbook. I list some conceivable new ways of using the Process Handbook here to motivate the development of PQL.

1.1.1 Process Redesign/Editorial Purposes

PQL will be a powerful instrument to support process redesign. With PQL, the user can find similar processes or processes with given attributes and relationships with other processes. Not only will the user be able to just search for processes, but he/she can also query for processes that are similar in structure to the current process but better in the specified ways. For example, PQL can help answer questions like, "What are processes that achieve purpose $\mathcal{Z}$ but uses resources $\mathcal{X}$ that are less costly than $\mathcal{Y}$?"

PQL also helps the editorial process of the Process Handbook. The administrator might be interested in all the leaves of the process tree (processes which have no specialization or decomposition), all dangling ports (ports that lead to nowhere), all the bundles that are without tradeoff tables, or all connected but mismatched ports (e.g., where some port attributes are not compatible.)

1.1.2 Dynamic Folders/Sharable Resources

A folder is used to store links to different processes that might be of interest to the user. Like references, a folder contains a group of pointers to other processes that are similar in some aspect. Currently, folders contain static links to processes and are updated by Process Handbook administrators when the database changes. PQL allows us to conveniently define dynamic folders which are actually "canned" queries; the content of the folder is the result of running the query on the Process Handbook at the time the folder is being examined. These folders are dynamic because their contents are based on a query and reflect the current state of the database. Minimal administration is needed to keep folders up to date. The challenge becomes specifying the predicate that determines if a process belongs in the folder or not in the form of PQL.

PQL can also be used to compactly describe shared resources used for the purpose of collaborative work or as a compact way of highlighting a particular kind of content. For example, if a user is working with a set of processes and wants to collaborate with his co-worker, he can either tell his co-worker to look for such and such processes on his own or he can send his co-worker a list of all the processes in his workspace. Alternatively, the user can compactly send his co-worker a query that can expand into the set of processes. What PQL thus brings to the table for both dynamic folders and sharable resources is a kind of late binding mechanism.

1.1.3 High Level Query Interfaces

Lastly, we need a high-level query language for specifying complex relationships and finding sets of processes that satisfy those relationships. Other than just having a formal language for querying the Process Handbook, many creative high-level query interfaces can be build to serve different end-user needs. One useful query interface might be, for example, a graphical query tool that allows the user to construct visually the process structure that he/she would like to search for in the database. Other high-level query interfaces can be more ad hoc to a specific purpose. These can be pre-assembled queries that the user needs only to fill in a couple of blanks or can even drag and drop a visual process structure into the query form blanks. Queries can be tailored to highly specialized situations for speed and convenience.

1.2 Queries we would like to make

There are many applications for PQL, but the base question involves what kind of queries can be expressed. Therefore, for the rest of this thesis, we will focus on the important questions of: how to express complex relationships of processes and how to find sets of processes that satisfy those relationships. In order to help us stay focused throughout this long thesis, I will pose a couple of scenario problems here that we should keep in mind for the rest of this thesis. By the end, you should be able to answer these questions or more specialized versions of it by using PQL.

1.2.1 The Internet Consultant's Query

You are a management consultant, who is hired to help a firm re-design some of its processes. You believe that the firm can cut cost and be more efficient if they move some of their processes on to the Internet. The firm is currently involved in a retail business and you want to look into ways to move some of their sales processes onto the Internet. In particular, you want to know "What are the different ways of selling that involve the Internet and that are generally more cost effective than their non-Internet counterparts?" Assume that you have identified which sales processes the firm should move to the Internet, now you are interested in how to market the firm's products for the new sales strategy. Specifically, you are interested in the following question: "What are the different ways to identify potential customers if you want to sell over the Internet."

1.2.2 The Operations Manager's Query

You are an operations manager, looking over the firm's sales processes. You have just read a report that DARPA is changing some of its policies and requirements on how it purchases goods from suppliers. DARPA being one of your firm's customers, you want to assess the impact of the DARPA policy changes on your firm's sales processes. You are interested in the following question: "What are all the sales processes that involve dealing with DARPA?"

1.2.3 The Human Resources's Query

You are a new member of the human resources department. You have just been assigned the responsibility of filling a number of positions in a short amount of time. Being new to the job, you have no prior experience in how to proceed. In particular, the different ways to hire human resources[1] and the appropriate ways of "identifying

[1]Human resources here refers to any type of human capital as in the sense of the word "resources". Please do not confuse it with human resources department mentioned earlier in the same paragraph which refers to the particular part of a firm that is responsible for hiring.

sources" for the different ways of hire human resources. In this case, you want to find out which ways "identifying sources" are appropriate for which types of hires. You are interested in the question: "What are all the ways of identifying sources that are used by different types of hiring?"

1.3 Organization of This Thesis

The semantic design of PQL is a large effort. It should involve constant experimentation by the programmer and feedback from the user. Only through the user, we can know what kinds of queries are most useful and what kind of syntax allows for easy expression of those queries. But through the programmer, we learn what kind of semantics of PQL can be implemented efficiently so that the user can get results in reasonable amount of time. This research will only experiment with a very limited subset of PQL semantics. In fact, our strategy is somewhat depth-first; we design a subset of PQL and we implement those parts immediately and experiment with it before moving back to design more parts or to modify existing parts.

Before entering into the discussion of PQL, some familiarity in the Process Handbook is necessary. In particular, the reader needs some background in the process model that we use. A general understanding of the Process Handbook architecture that we have developed so far is a plus but not necessary. Readers who are already familiar with the Process Handbook can skip to chapter 3, Current Methods for Querying the Process Handbook. Chapter 2, The Modeling of Organizational Processes, introduces some important concepts of our process model. Specifically, chapter 2 discusses the two dimensionality of the process model (decomposition and specialization) and the concept of dependencies and managing of dependencies. The reader is also encouraged to read appendix A, The Process Handbook Architecture, to become familiar with the framework in which we are building PQL. Appendix A describes how the concepts in chapter 2 are put into practice. It surveys each part of the three-tier client/server architecture: database schema, Object API middle ware, and all the applications that runs on top of the object API. Appendix A also serves

as a good summary on the current state of the development and research. It should be useful to someone who is new to the research center and wants a quick grasp on how everything fits together. Besides, many changes and new ideas have been implemented in the Process Handbook, both by myself [2] and by other researchers, since papers had been last published on the Process Handbook, which makes this appendix section worthwhile.

With backgrounds out of the way, chapter 3, Current Methods for Querying the Process Handbook, describes the current techniques available for querying the Process Handbook. The chapter discusses three query techniques (navigation, keyword search, and SQL query) and criticizes their problems, thus motivating the need for a true process query language (PQL). Chapter 4, Toward a Process Query Language (PQL), starts the discussion on the design of PQL. The first part of chapter 4 discusses the design issues of PQL such as design goals, architecture, and searching algorithms used. The second part of the chapter explains the syntax of the subset of PQL that is implemented and shows examples of how to express queries in this language. Finally, the chapter shows how the complex queries that we have posed in the previous section can be expressed using PQL and what kind of answers the current Process Handbook returns. Chapter 5, Evaluation, reviews our design goal and analyzes to what extent those goals are achieved. Lastly, chapter 6, Future Work, discusses ideas that are left unimplemented because of time constraints. It explains three dimensions of PQL in which further research are worthwhile.

The appendices contain material that is relevant to the thesis but don't fit into the flow of the paper. I have already mentioned appendix A, The Process Handbook Architecture, which describes the latest architecture and look and feel of the Process Handbook. Appendix B, Search Algorithm, discusses the search algorithms that I have implemented in PQL. Appendix C, PQL Implementation Notes, presents the design and implementation of PQL in more detail. This implementation notes is written mainly for future researchers who will be modifying or extending the imple-

[2] On the extension and the new look and feel of the Web Process Handbook (The 1997 sponsors meeting version)

mentation of PQL.

Chapter 2

Background: The Modeling of Organizational Processes

The first step to understanding and redesigning processes is to develop a set of primitives that can express the features of a process that are most useful to the user trying to understand or redesign the process. The Process Handbook model is such a set of primitives for describing processes or activities (the term process and activity will be used interchangeably in this thesis) in a concise and general schema. The expressiveness of our process representation model is the result of two major ideas: (1) the two-dimensional approach (decomposition and specialization) to process analysis, and (2) concepts of managing dependencies from coordination theory (see [6] for more detail).

2.1 Decomposition and Specialization

The process compass (figure 2-1) illustrates the two dimensions that our process model spans. The North-South direction represents the dimension of composition and decomposition of an activity while the East-West direction represents the dimension of generalization and specialization of an activity.

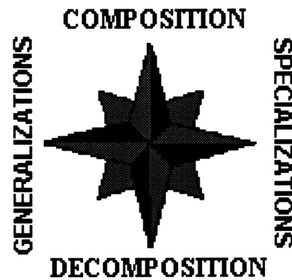


Figure 2-1: The Process Compass (Adapted from Process Handbook Web UI)

2.1.1 Decomposition

Decomposition is the process of breaking down (or "decomposing") an activity into sub-activities. The decomposition of activities allows the user to analyze any activity at the right level of abstraction (or detail). While a manager might be interested only in the high level activities, an engineer will want to see a much lower level decomposition of that activity. Theoretically, any activity can be decomposed infinitely, but in practice a process will only be decomposed to a level that is deemed useful or desirable to the user. Moving downward, the user can see more and more detailed decomposition (sub-activities) of a given process while moving upwards, the user can see how the given process is used (where used) in the composition of other larger processes.

Figure 2-2 illustrates one way the activity Sell Product can be decomposed into its sub-activities. In this example, Sell Product is decomposed into sub-activities of Identify Potential Customer, Inform Potential Customers, Obtain Order, Deliver Product and Receive Payment. Each of those sub-activities, in turn, can be further decomposed.

2.1.2 Specialization

In addition to decomposing processes into sub-activities, the Process Handbook model also includes the concept of specialization borrowed from the inheritance model of object-oriented programming. Specialization is a way to differentiate (or specialize)

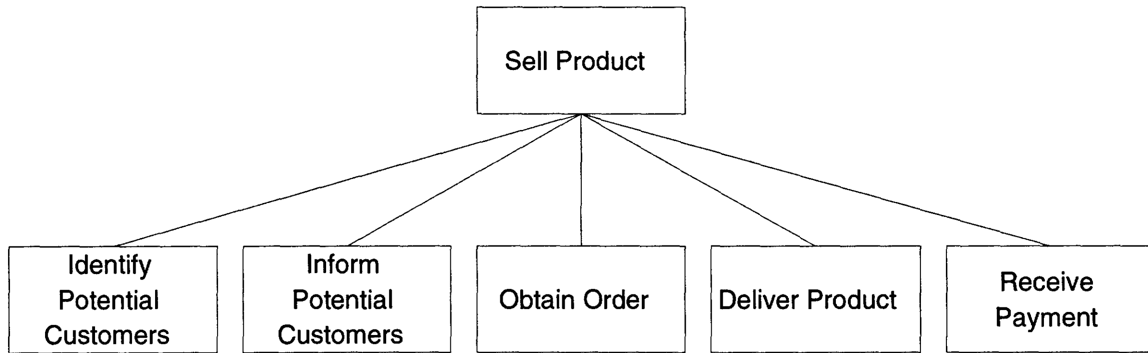


Figure 2-2: The decomposition of Sell Product (Adapted from [6])

a process into various types of processes that are more tailored to some subset of the original domain. Therefore, "a sub-activity represents a part of a process; a specialization represents 'a way' of doing the process."[6] Our generic process *Sell Product* can be specialized into processes like *Sell by Mail Order* and *Sell in Retail Store.* Applying specialization to process results in a hierarchical network of processes where the generic processes are on top and the more specialized processes at the bottom. This method becomes powerful when we incorporate the concept of inheritance into the system; specialized processes automatically inherit the decomposition and other attributes of their "parent" process.

Figure 2-3 illustrates both the decomposition and specialization of the process *Sell Product* simultaneously. The shaded sub-activities are those inherited without change from their parent. Specialized processes add to and/or change the decompositions they inherit. In our example, *Sell by Mail Order* inherits the sub-activities *Deliver Product* and *Receive payment* from *Sell product* without change. However, *Identify potential customers* is replaced by the more specialized activity of *Obtain Mailing Lists.* Similarly, *Inform Potential Customers* is specialized to *Mail Ads to Mailing Lists* and *Obtain order* is specialized to *Receive order by mail.*

As a comparison, in *Sell in Retail Store, Identify Potential Customers* is specialized to *Attract Customers to Store.* The idea of *Sell by Mail Order* and *Sell in Retail Store* are two ways to *Sell Product* is clearly and straightforwardly represented in such a

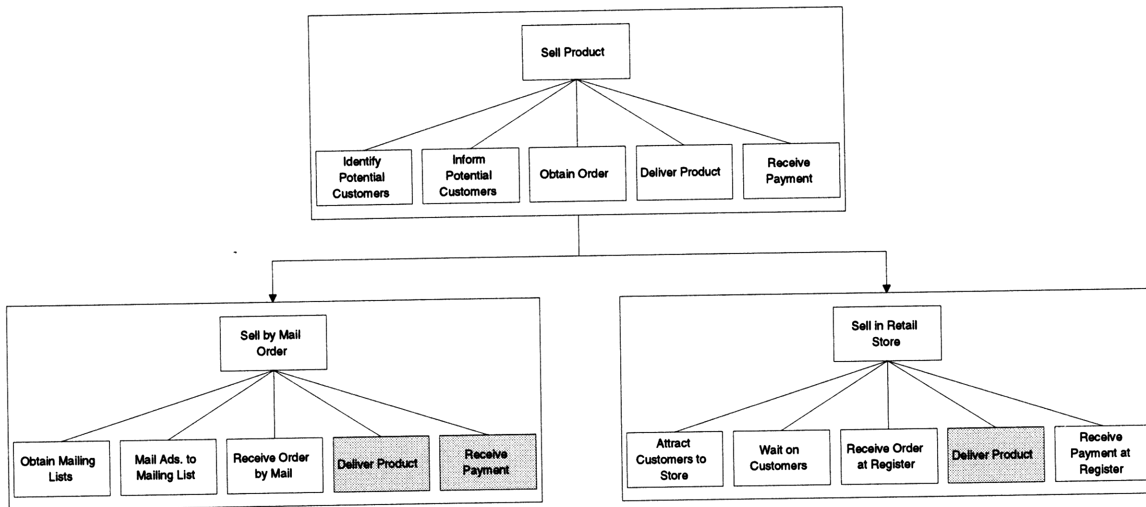


Figure 2-3: The representation of a generic sales process (*Sale Product*) and its specializations (*Sell by Mail Order* and *Sell in Retail Store*). (Adapted from [6])

model. We don't have to stop there; any sub-activities can be further decomposed or specialized.

There is a subtle difference between inheritance in our model and that of traditional object-oriented programming. Traditional object-oriented programming is a paradigm of inheriting down a hierarchy of nouns (increasingly specialized objects which may have associated with them actions), whereas our process model is a paradigm of inheriting down a hierarchy of verbs (increasingly specialized actions which may have associated with them objects).

Bundles and Trade-off Matrices

Bundles are developed to organize specializations into groups of related alternatives. Figure 2-4 shows how specializations of the process *Sell something* are grouped into two bundles: [Sell how?][1] and [Sell what?]. The [Sell how?] bundle of specializations for *Sell something* groups processes of how the sale is made: direct sales, retail store, mail order, or over the Internet. The [Sell what?] bundle of specializations groups processes that are about what is being sold: beer, telecom services, academic supplies,

[1]Bundles are expressed with [ ] by convention.

automotive components or financial services.

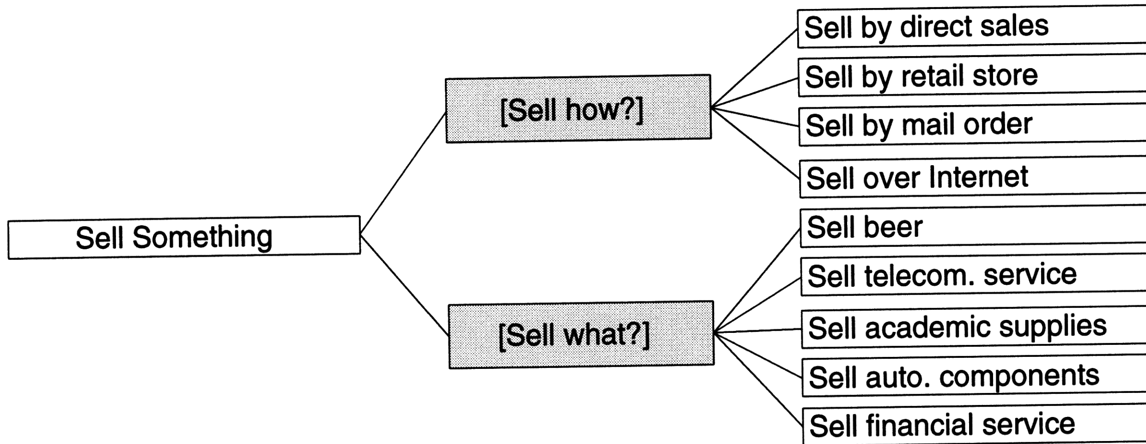


Figure 2-4: Specializations of the activity Sell Something grouped by bundles [Sell how?] and [Sell what?]. (Adapted from [6])

Bundles are technically not part of the specialization hierarchy; they are added only to help organize the specialization hierarchy in a more useful way. In our process model, bundles are used in two ways: grouping comparable alternatives of specializations and comparing them on multiple dimensions in a tradeoff matrix. Figure 2-5 shows a "tradeoff matrix" where specializations of the [Sell how?] bundle are being compared in the dimensions of cost, time, and quality. Since our specialization processes are still very generic processes, comparisons are made qualitatively. But if our specialization processes are sufficiently detailed, the "tradeoff matrix" can contain very quantitative performance metrics. The information in tradeoff matrices can be rough estimates by managers or they can be the result of intensive research or studies.

Combining decomposition and specialization together with the concept of bundles and tradeoff matrices greatly increase the expressive power of our process model. A user can study the various levels of abstraction (or decomposition) which enhances the understanding of existing processes. At the same time, a user can take existing processes and tailor (or specialize) them to meet specific needs which enhances the generation of new processes. Inheritance greatly cuts down on the turn around time of generating new processes and enhances ease of modifying existing processes which

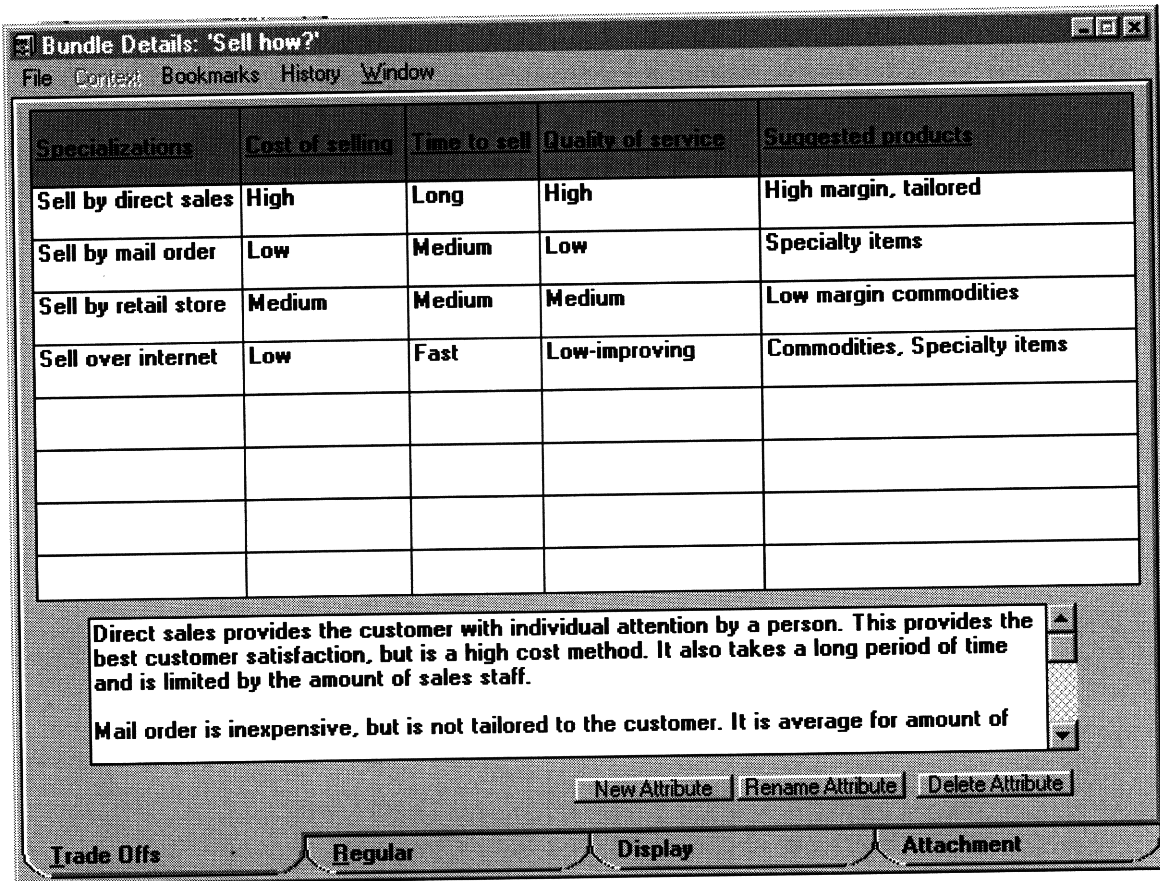


Figure 2-5: A tradeoff matrix showing rough estimate of advantages and disadvantages of different specializations for the generic sales process. (Adapted from [6])

other processes depends on.

2.2 Dependencies and Coordination

A good description of a process cannot be complete with just a set of its decompositions; it must also specify the dependencies among its decompositions. To illustrate dependencies, I will use the classic process, Make Coffee. The reason is that Make Coffee is simpler (two sub-activities) than Sell Product so diagrams would be less cluttered. Figure 2-6 shows that Make Coffee can be decomposed into two sub-activities: Fetch Coffee Beans and Brew Coffee. However, we need to express the idea that we need to fetch coffee beans before we can brew coffee as shown by the extra flow depen-

dency in figure 2-6. Actually, the dependency needs to specify three things: (1) the beans have to be fetched before the coffee can be brewed (prerequisite constraint), (2) the beans have to be transported to the place where the coffee will be brewed (accessibility constraint), and (3) the brewing mechanism should be able to use the coffee beans that are fetched (usability constraint). Our process model uses a set of primitives (dependencies, ports, connectors, and managing activities) to express such dependencies among activities.

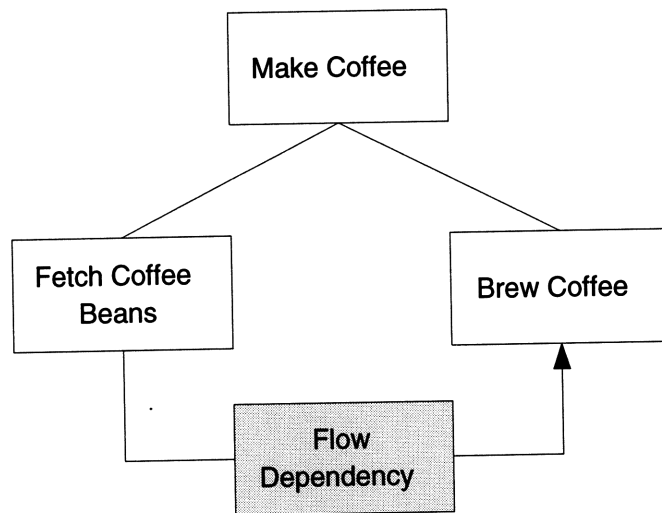


Figure 2-6: Decomposition and dependency (Adapted from [4])

2.2.1 Dependencies

Ideas for modeling dependencies of activities are borrowed from coordination theory. Coordination can be defined as "managing dependencies among activities."[6] The definition entails two steps: (1) identifying the kind of dependency and (2) choosing coordination process that can manage such dependency. There are many different kinds of dependencies and there are many alternative coordination processes that can manage a given dependency.

The three basic kinds of dependencies are flow, sharing, and fit. When a resource produced by one activity is consumed by another activity, we have a flow dependency.

Our Make Coffee example exhibits a flow dependency between Fetch Coffee Bean and Brew Coffee. A sharing dependency occurs when a resource is consumed by multiple activities (e.g., when two activities need to use the same machine). A fit dependency arises when multiple activities produce a single resource (e.g., when components manufactured independently must fit together). More complicated dependencies among activities can arise when the number of processes involved increases.

Having identified the basic dependencies, now we turn to coordination processes that can manage those dependencies. For example, a sharing dependency can be managed by a variety of coordination processes. In a supermarket, the sharing dependency of a cashier is managed by a queue (or "first come/first serve" mechanism). In a work place, sharing dependency can be managed by priority order (process more important tasks first) or managerial decision (when two tasks appear to be of same priority). In school, two athletic activities wanting to use the same gym can be managed by budgeting (having pre-assigned time slots). In business, the owner of a resource might even have a market-like bidding system (sell time using resources to those who are willing to pay the most for it). These few generic processes can lead to many different specializations. In fact, the coordination processes that manage dependencies are themselves first class citizens, having decompositions and specializations and dependencies of their own.

2.2.2 Ports and Connectors

Activities and dependencies interact with each other through ports (input and output interfaces). In the Make Coffee example, the activity Fetch Coffee Beans would have a port that represents the output (production) for coffee beans that have been fetched. On the other end, the process Brew Coffee would have a port that represents the input (consumption) of coffee beans. Figure 2-7 shows the addition of the dependencies representation to the decomposition of Make Coffee. As you can see, dependencies and ports are treated as part of the decomposition just like sub-activities are. Decomposition Relations (DR) connect a parent activity to its decompositions (sub-activities, dependencies and ports).

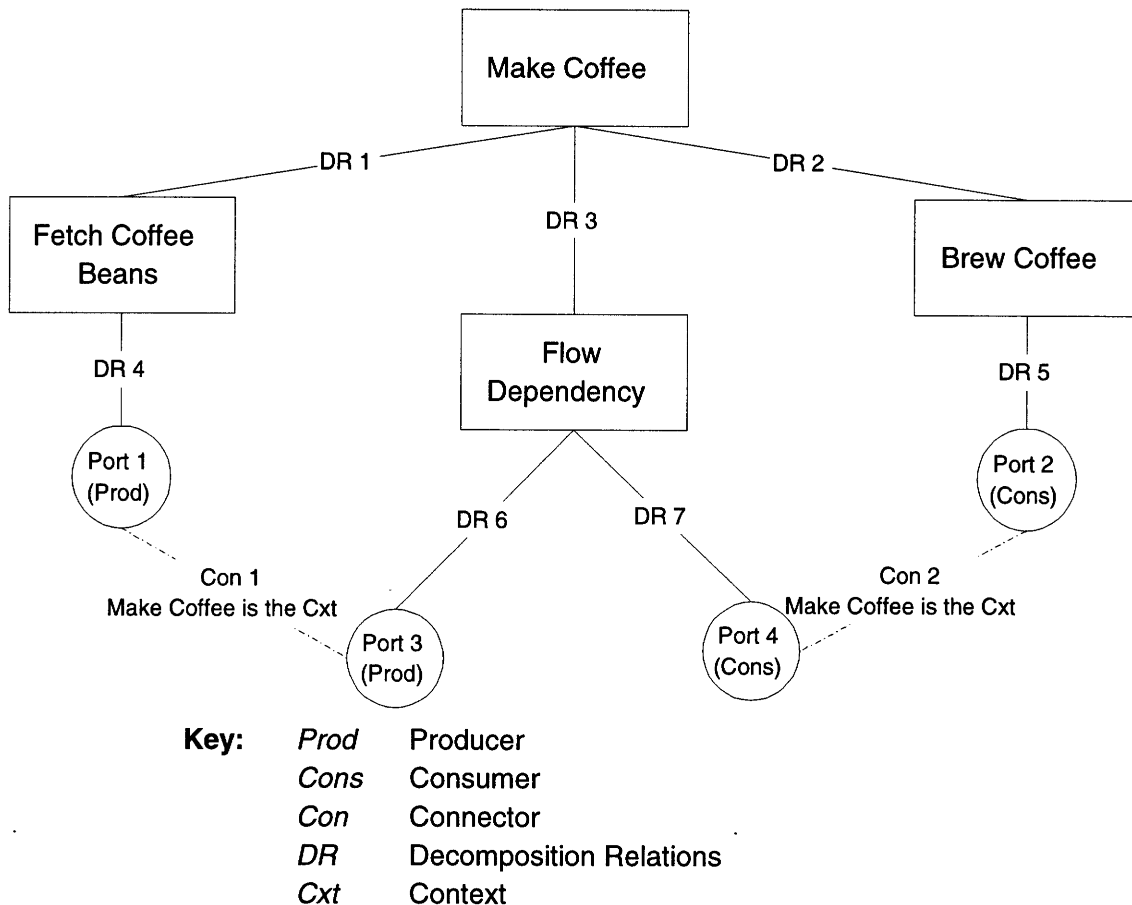


Figure 2-7: Decomposition of *Make Coffee* with dependencies expressed. (Adapted from [4])

Another type of relation is a Connector, which connects ports of different activities and dependencies. There are two kinds of connectors: horizontal connectors and vertical connectors. Horizontal connectors connect ports of siblings in decomposition. In figure 2-8, connector Con 1 is a horizontal connector since it connects Port 1 of Fetch Coffee Beans and Port 3 of Flow Dependency which are siblings in the decomposition of Make Coffee. Vertical connectors connect ports of an activity to ports of sub-activities in its decomposition. Vertical connectors are important instruments when we have multiple level of dependencies and when we introduce managing activities. Figure 2-8 shows two levels of decomposition of some activity A.

Activity A has only one dependency Depe 1. In this case, activity B is decomposed

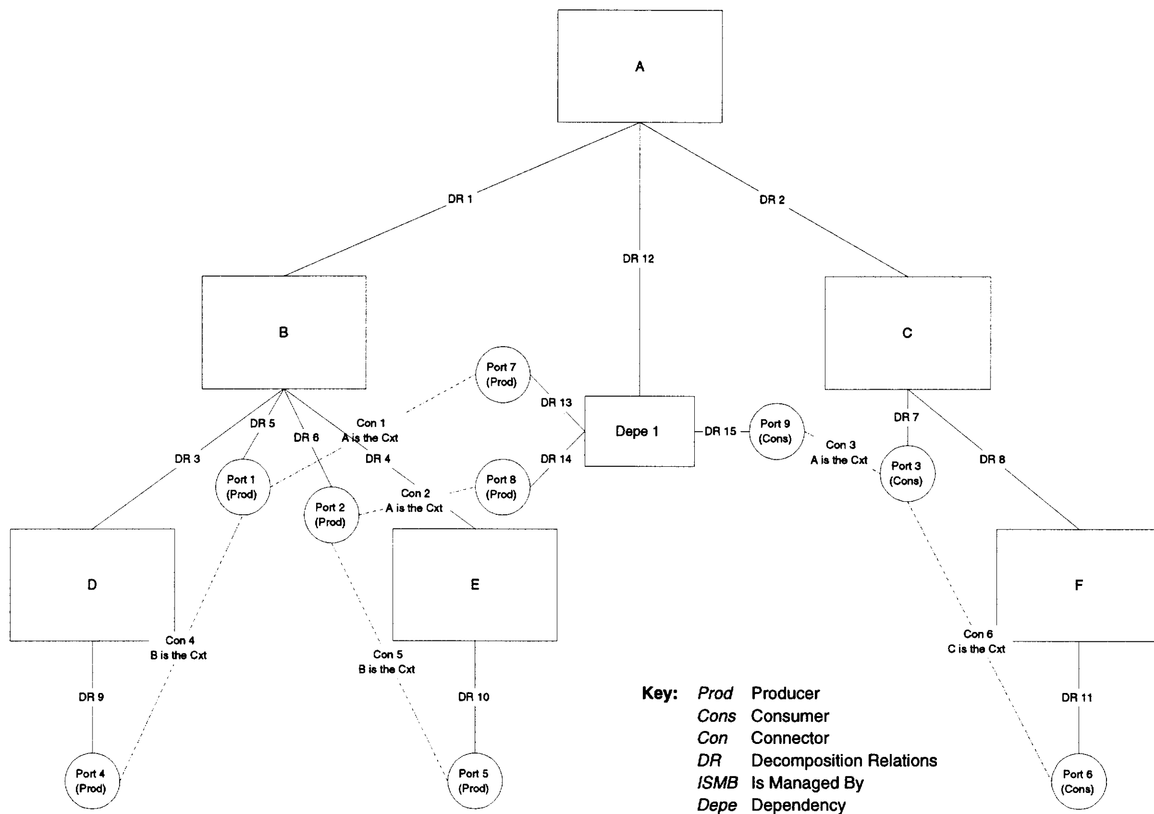


Figure 2-8: The semantics of multiple levels of decomposition, vertical connectors and dependency with multiple producers and a single consumer. (Adapted from [4])

into activities D and E and each is a producer to Depe 1. Activities D and E have producer ports Port 4 and Port 5 respectively. At the same time, on the abstraction level of activity B, two producer ports, Port 1 and Port 2, are needed to represent the output of B, whether we know about its decomposition (D and E) or not. It is obvious that Port 1 and Port 4 represent the same port in reality as well as Port 2 and Port 5; they are only on different levels of abstraction. Therefore, they are connected by vertical connectors to show that they are the same ports. In the diagram, Port 1 and Port 4 are connected by Con 4 and Port 2 and Port 5 are connected by Con 5. Vertical connectors are used to show which ports of the sub-activities correspond to which ports of its parent. Connectors exist only in the context of an activity. Horizontal connectors have the decomposition parent of the dependency as their context. Vertical connectors have the parent activity as their context.

2.2.3 Managing Activities

Dependencies are managed by managing activities. In our Process Handbook model, the user has the ability to choose any process from a repository of managing activities that can manage a given dependency. Figure 2-9 shows the results of adding a managing activity M for the dependency Depe 1 in process A. The original schema (decomposition relations, connectors, and activities) remains unchanged and they are not shown in the diagram to avoid cluttering. What are new are the additional dependencies, ports, connectors and decomposition relations (all drawn with thicker lines) created to interact with the managing activity M. As you can see from the figure 2-9 that dependency Depe 1 is decomposed into sub-dependencies Depe 2, Depe 3 and Depe 4. In general, the introduction of a managing activity can decompose any complicated dependency into a set of simple (one producer/one consumer) dependencies.

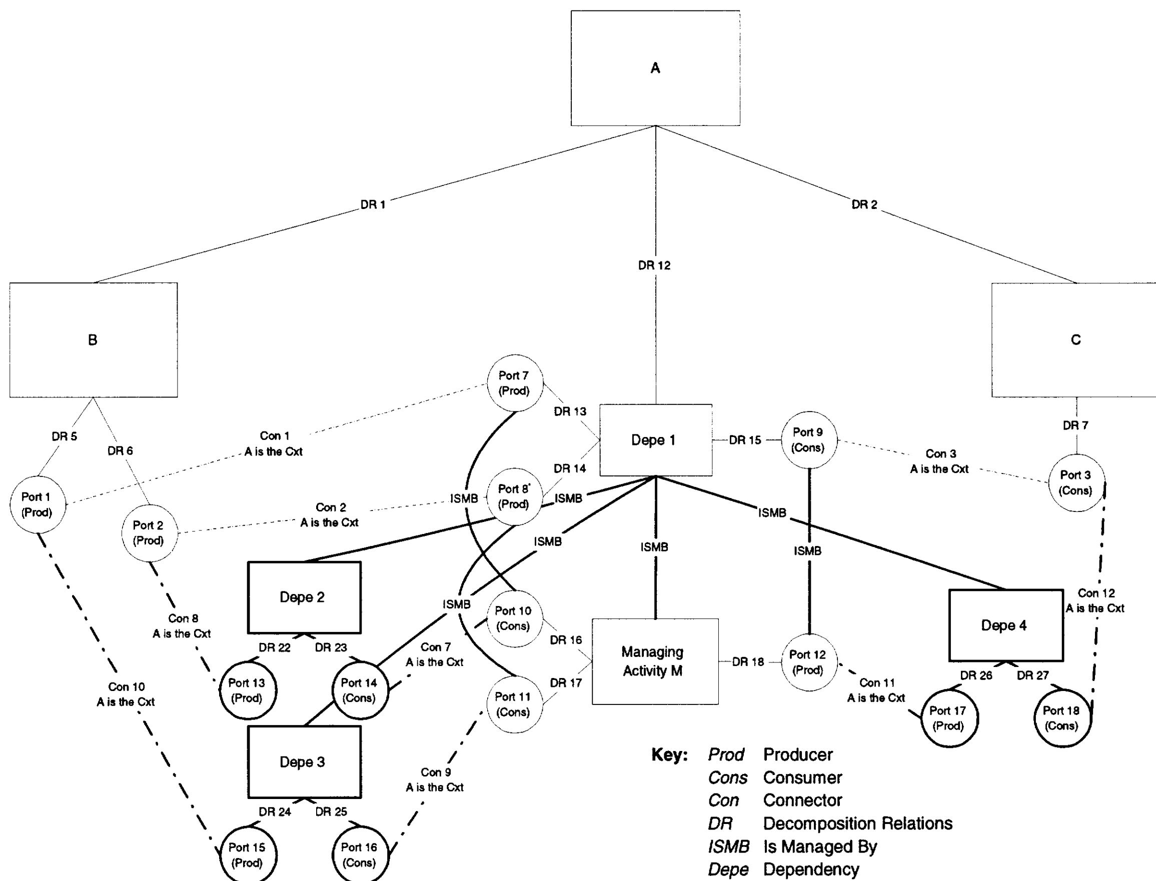


Figure 2-9: The changes to the decomposition of A after a managing activity is set for the dependency Depe 1. (Adapted from [4])

Chapter 3

Current Methods for Querying Process Handbook

Having discussed the vast amount of complex information stored in the Process Handbook in the previous chapter, it is now time to show how one can find what are needs in the Process Handbook. After all, a large amount of information is useful only when you can find what you want. The current system provides three simple methods for finding processes.

3.1 Find by Navigation

The first method of finding a process is just to navigate through the Process Handbook tree hierarchy (The term tree is not strictly correct here because the Process Handbook navigation hierarchy contains cycles). The Process Handbook contains a high-level framework of a small number of very generic activities, and then classifies all other activities as specializations of these high-level activities. The current version of this taxonomy has general activities like *Create*, *Destroy*, *Modify*, and *Preserve* at the top level. These generic processes are then further specialized down to all other activities in the Process Handbook.

This type of organization, categorizing processes as specializations of more generic abstract processes, is one type of hierarchy that users can use to navigate to a par-

ticular process and is located under Process Specialization Hierarchy. Figure 3-1 illustrates a path to the process *Make Telecommunications Service* in the process specialization hierarchy of the web process handbook. The path traversed involved starting from the high level generic process *Create*, navigating through bundles (e.g., [Create what?], [Make what thing?], [Make what service?]) and specializations (e.g., *Create Physical Item*, *Make Something*, *Make Service*) that gets more and more specialized, and finally arriving at *Make Telecommunications Service.*

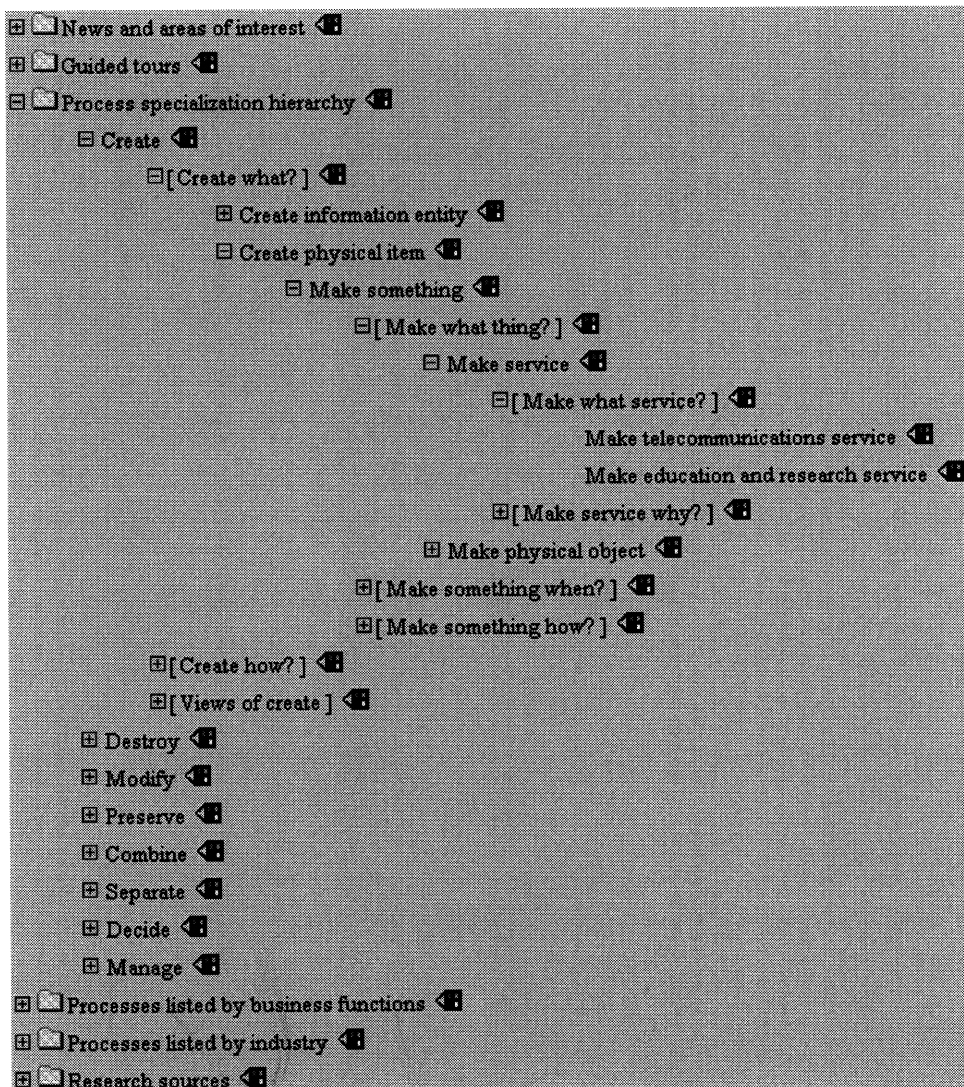


Figure 3-1: Process specialization hierarchy from the Web Process Handbook.

The process specialization hierarchy is only one type of organization that a user

can use to navigate to the destination process. The Process Handbook also organizes processes into other useful hierarchies such as *Processes listed by business functions* and *Processes listed by industry.* For example, under *Processes listed by business functions* we have categories of *Supply Chain Management, Marketing, Sales, Information Systems, Human Resources, Strategic Planning, Finance/Accounting, Manufacturing/Logistics* and *Engineering* which are a very different types of organization than the Process specialization hierarchy. Look back at figure A-5 to see where the process Sell something is located in the *Processes listed by business functions'* hierarchy.

Once we are focused on one process (e.g., *Sell Something*), we can also navigate in the four directions of the process compass (sub-activities, specializations, uses, and generalizations) with respect to the currently focused process (Sell something) as illustrated previously in figure A-5. Traveling in the sub-activities direction leads us to *Identify Potential Customers, Inform Potential Customers, Identify Potential Customers' Needs, Obtain order, Deliver the Thing,* and *Receive payment.* Traveling in the specialization direction leads us to specializations such as *Sell in Retail Store, Sell Electronically, Sell by Direct Marketing* and etc in the [Sell how?] bundle and *Sell Financial Service, Sell Telecommunication Service, Sell Beer* and etc in [Sell what?] bundle. Similarly, we can get to more general processes by traveling in the generalizations direction and processes that uses the current process in their decompositions by traveling in the uses direction.

3.2 Problems with Find by Navigation

Even though such hierarchical organizations provide a clear way to communicate how different processes are related through their ancestors, obviously they alone are not sufficient to serve as the method to finding processes. First, the organization with specializations and bundles may not be as intuitive to a user new to the process handbook. Second, as the database grows, the hierarchical structure will get deeper and deeper until it becomes very inconvenient to get to the targeted location. Navigation would become tediously slow when used in sparsely populated branches.

3.3 Search by Keyword

The second method of finding processes is the search by keyword capability in both the Windows and Web Process Handbook interfaces. This function provides a way to get to the targeted process directly without having to navigate through a path of related processes. The most common way of using this function is to search for a process by its name. Figure 3-2 shows how we can search for any process with *Sell over Internet* as part of its name and the results returned in the Web Process Handbook search interface.

As shown in figure 3-2, the search returned three results: a bundle [Sell over Internet how?] and two processes *Sell Over Internet* and *Sell Over Internet Mall.* We can also search for the keyword in a process's description and/or notes.

The Windows application of Process Handbook also provides a similar capability with the added feature of searching not just for an activity but bundles, connectors, dependencies and ports as well. Searching for connectors, ports and dependencies are useful in the dependency editor. Figure 3-3 shows the search interface from the Process Handbook application.

3.4 Problems with Keyword Search

Keyword search is currently the only querying method an end user has for finding processes. But the limitations of keyword search have prevented users from taking advantage of the rich semantics of our process model. The most common problem with keyword search is the failure to find what we were looking for when the information is in the database. This is called a false negative and it is often due to terminology mismatches: when we search for a process using a different terminology from what the database uses. For example, if we are interested in finding processes about on-line shopping, we might search the database with "cyber mall" as keywords on both entity name and description perhaps. The search would return with no matches found in the database. Perhaps we might try again with "online-shopping" as keyword. Again,

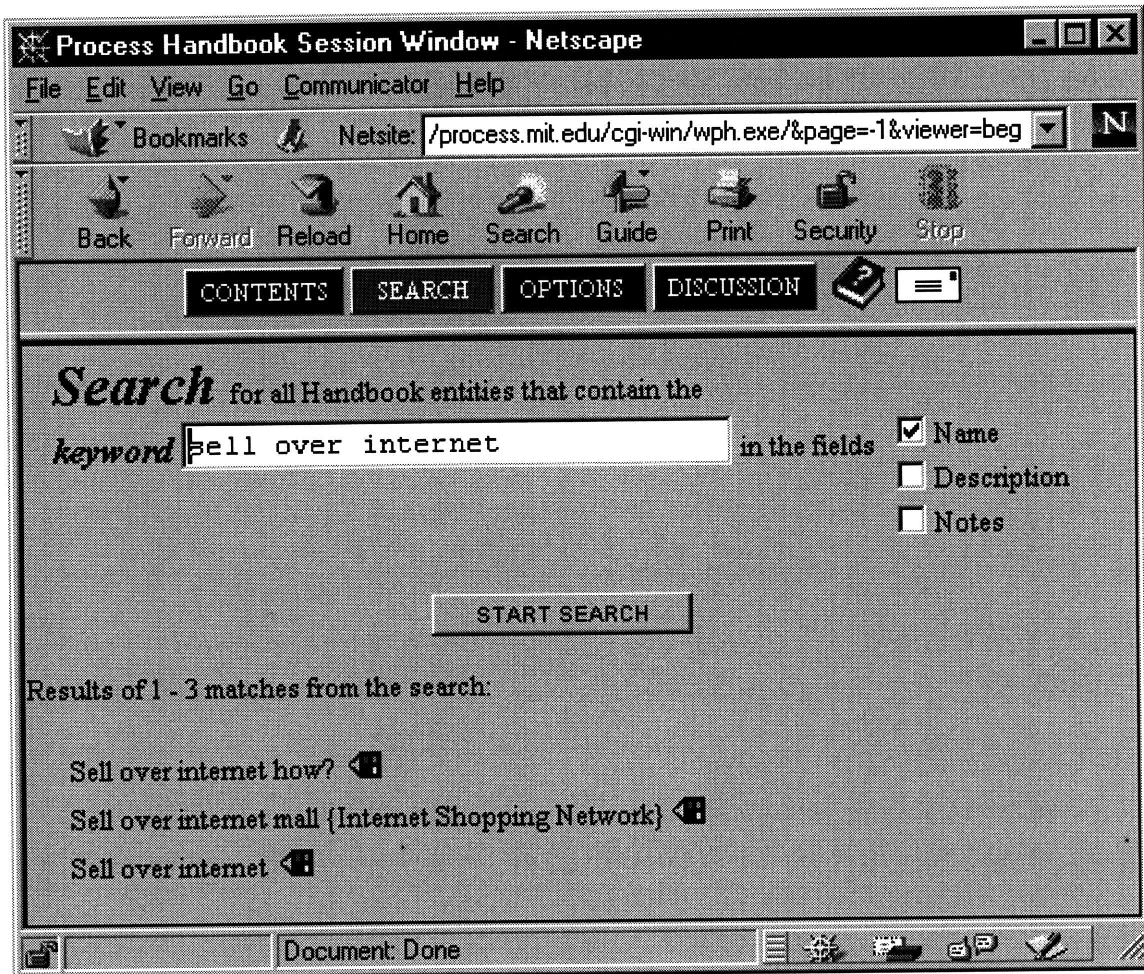


Figure 3-2: Search interface from Web Process Handbook

no matches were found. We would probably give up, thinking that there is no such process in the database, when in fact, there is a process called *Sell Over Internet Mall*, which is exactly what we were looking for. But we failed to find it because of the differences between our terminology and that of the database.

Another similar error is false positive, the success of finding something that matches our keywords but they are not what we meant to look for. For example, if a firm is interested in hiring someone to work in the human resources department (recruiting/hiring department), they might search the Process Handbook for "Hire human resources". This time we would get a perfect match; there is a process called *Hire Human Resources* in the database. However, the *Hire Human Resources* in the

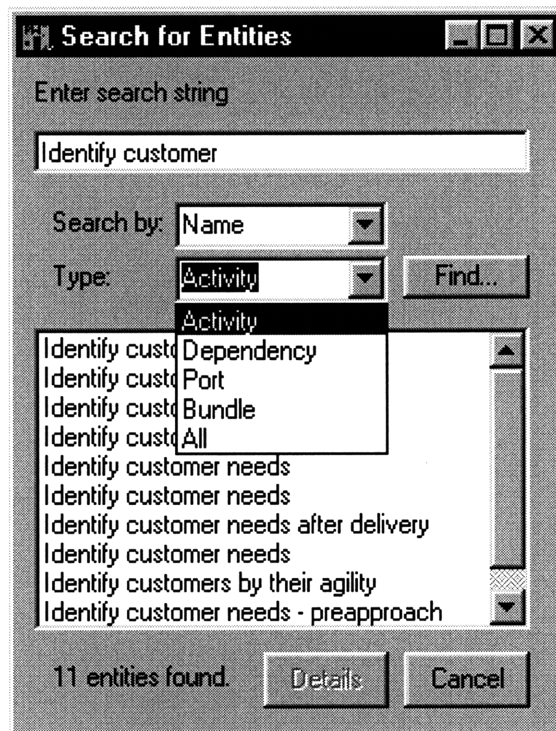


Figure 3-3: Search interface from the Windows application of the Process Handbook

process database refers to a much more general hiring process. "Human resources" in the Process Handbook refers to any kinds of human workers (resources), not just the specific people who do recruiting in a firm. Here, terminology mismatch results in false positives. False positives also happen when a search returns far too many matches, thus defeating the purpose of the search.

We would like to have a query language that can make use of the information and structures already inherent in the Process Handbook. PQL should give the user a vocabulary to describe a process that is beyond just names and stored descriptions. In particular, the power of the Process Handbook comes from the modeling of the relationships among processes; a standalone process is often of little interest or use. PQL must be able to exploit this power and give the user the ability to describe or search for a process based on its relationship to other processes. For example, to find a process on on-line shopping, the user should be able to describe such process as a type (specialization) of selling that uses the Internet. Here, we want to find a new

process based on the relationship that it has with two other processes. This way, we are not affected by the use of different terminology.

3.5 Query by SQL

The most powerful query method available is still to query the database with SQL. The Process Handbook is designed on a relational database, so naturally, everything that we would want to query can be expressed by SQL. However, this query method is only available to the programmers and not the end users. Before the advent of Object API[1] all applications accessed the database directly using SQL; it was the only way to extract information from the Process database. After the implementation of Object API, most applications now access the database through the Object API. However, the Object API did not provide any query mechanism. Therefore, complex queries must still be done with SQL directly on the database.

SQL statements consists of three clauses: select, from, and where. The select clause is the projection operation of relational algebra. It is used to filter the results so that only those attributes desired in the result of the query are listed. The from clause is Cartesian product operation of the relational algebra. It specifies the relations to be searched in the query expression. The where clause is where you specify the selection predicate of the relational algebra. The predicate is a Boolean expression involving the attributes of the relations that is listed in the from clause. An SQL statement has the form:

Select $A_1, A_2, \ldots, A_n$
From $r_1, r_2, \ldots, r_m$
Where p

In order to query the Process Handbook database directly with SQL, the user must know the database schema (figure A-1) well. The user must also understand

[1]The Object API abstracts the database representation of processes and exports an object-oriented view of processes. See Appendix A for more detail.

the details of how processes and relations are represented in relational table form. The following SQL statement finds all the processes that are in the decomposition of *Sell Over Internet* which is identified by the entity ID, 3122.

Select ent_id
From $Entity$, $Relation$
Where $Relation.rel_start_ent_id = 3122$
$and\ Relation.rel_end_ent_id = Entity.ent_id$
$and\ Relation.rel_type = 1$
$and\ Entity.ent_type = 1$
$and\ Relation.rel_deleted = False$
$and\ Entity.ent_deleted = False$

The strategy of the above SQL statement is to go through the relation table and select those entries that have the process ID 3122 as the beginning of a decomposition relation. Then it verifies in the entity table that the processes that are at the other end of these decomposition relations are of entity type.

3.6 Problems with Query by SQL

While SQL is a powerful relational query language, it does not serve well as our process query language. The main reason is the mismatch in abstraction. The process model is built on top of a relational database. Therefore, we would like to use a language that is efficient in describing processes and relations between processes, not a language that is efficient in dealing with the underlying representation of our process model. Furthermore, querying at the process representation level defeats the very purpose of our process model.

Because of the above mentioned abstraction mismatch, querying the process database using SQL becomes unmanageably complex even for a research scientist or Process Handbook programmer. We have already seen the SQL statement that gets the decompositions of a given process (This operation has already been implemented as

"GetDecomposition" method for the entity object in Object API). This operation is a very simple query. Now imagine that we want not just the immediate decompositions of process A, but all the processes that A can be ultimately be decomposed into. Such "deep" or "recursive" decomposition method can be easily implemented with a little help of a general programming language, but would become unnecessary complex and arcane if expressed as one SQL query. If you still think the "recursive" decomposition query is not that bad, consider expressing such query: "What are all the specializations of *Identify Potential Customers* that are used by *Sell over Internet* ?"

Querying directly into the process database is also bad abstraction practice, especially when it is exercised by third party applications. Once the underlying representation of the process model is exposed, the underlying representation becomes difficult to change or to improve. Switching database vendor (e.g., from MS Access to Oracle) or database technology (e.g., from relational database to object oriented database) would be impossible in the future without making many changes to existing applications. Object API is implemented precisely to provide such an abstraction between the process representation and the process model. It is only natural that we should create a query language that honors this abstraction as well.

We therefore need a query language appropriate for expressing processes and relationships among processes. The language has to be flexible, compact and convenient to use. Where appropriate, the language should optimize convenience and efficiency for the frequently used queries, even at the expense of complicating some more exotic queries. We discuss our design for such a language in the next chapter.

Chapter 4

Toward a Process Query Language (PQL)

What is in the database is only as useful as what you can get out of it. And what you can get out of a database depends on what you can ask of the database. Having pointed out the weaknesses of all the previously implemented ways of extracting information from the Process Handbook Database in the last chapter, it is now the appropriate time to discuss the design of a querying language that can serve our purposes better. We will discuss PQL in three parts: design and implementation, language syntax, and usage.

4.1 The Design of PQL

We will first look at the design goals of PQL. This section discusses the design decisions that were taken and how PQL fits into the overall Process Handbook architecture (described in appendix A). The core search engine of PQL will be described.

4.1.1 Design Goals and Issues

When the idea of PQL was first tossed around in the Coordination Science center, we were undecided about many important aspects of the new query language. Most of

the major issues involved important tradeoffs and have strong impact on the future direction of PQL. For example, what is the end-user interface or syntax that PQL should have? Should PQL be simple though incomplete or more complex and complete? We can design PQL to be especially convenient for the high frequency queries or alternatively, we can make it a powerful, though less convenient, language that can express any imaginable query on the Process Handbook? There are many difficult design decisions to be made but we believe that there are a few simple goals that we agreed on from the start. These are,

1. High level. Reasonable queries should be easily and concisely expressed at the end user level. The syntax should be easy to learn for Process Handbook users. It should be able to express a reasonable set of queries in a simple way. Simplicity and convenience often give rise to new queries that are otherwise impractical or inconceivable.

2. Database independent. How to query the process database should not depend on which database is used. PQL should not expose any of the process representation details.

3. Increased precision. PQL should reduce false negatives and false positives.[1]

4. The UI for defining queries should be a straightforward enhancement of editing metaphors already in use. The UI should be flexible and extensible.

5. Optimizable. The system should be able to find a good strategy to optimize user queries.

Another open issue is where is the correct place (the correct abstraction level) for PQL in the Process Handbook architecture (see appendix A). Who will be the client of PQL? End-user clients can be business people, CCS research scientists or programmers. Each of these scenarios calls for a different optimal design. Both Web

[1]False negatives and false positives are discussed in chapter 3.4.

and Windows version of Process Handbook can be clients of PQL, making use of "on the spot" queries to implement their dynamic folder capabilities.

Lastly, we need to consider what kind of data should PQL return as a result of a query. One approach is to return query results as objects which fits into the Object API middle ware perfectly. Another approach would be to return results in ad hoc protocols established with different end user applications.

Unfortunately, we are not ready to finalize many important decisions. The idea of PQL is relatively new. While there are many different ideas of possible applications being tossed around, the role and place for PQL in the Process Handbook architecture is still unclear. We simply do not have enough experience, end-user requirements, usage patterns, and performance benchmark upon which to base our design decisions. The goal of our prototype is to experiment with the syntax and implementation of such a language, to get user feedback and to collect a wide range of sample queries that users would like PQL to be able to handle. Because of this situation, we designed PQL using a two-pass approach that allows for maximum flexibility and postpones many design decisions to the near future.

4.1.2 PQL Architecture

PQL can conceivably be implemented at two places: above the Object API or at the Object API level. In the former design, PQL would access the process database through the Object API, manipulating processes as objects. This design is similar to the Object-Oriented Query Languages (OOQL), which are currently under extensive research by M.H. Scholl and H.-J. Schek in the COCOON Project [10]. The latter design calls for direct access to the process database using the database native query language (SQL in this case). This approach is more efficient but PQL implementation must be modified if the underlying database is changed.[2]

We choose the latter design on the bases of efficiency. Figure 4-1 illustrates the PQL module with respect to the overall PH architecture. The architecture diagram

[2]Actually only the database access methods need to be changed, which ought to be a small part in a good design.

illustrates three main points. First, PQL engine is implemented at the same level as the PH Object API, accessing the database directly using SQL. Second, PQL and PH Object API can make use of eachother's capabilities. For example, PH Object API can export an object version of the search capabilities. API functions can call PQL to perform the search. When PQL returns the results, the API function can instantiate the PQL results into entity objects for the Object API client. Third, higher level applications can directly interact with PQL or PH Object API if ad hoc protocol or efficiency is desired.

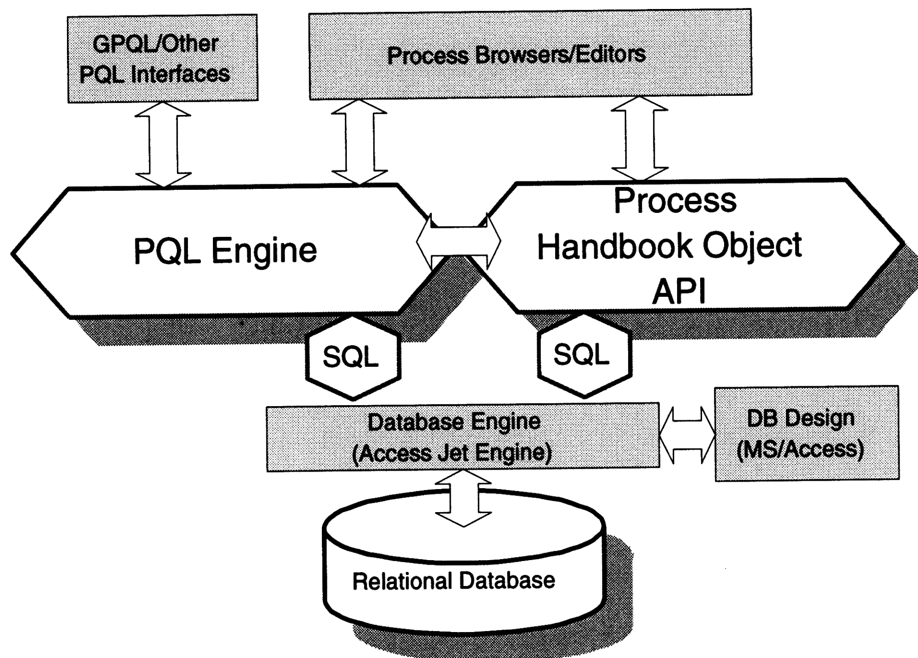


Figure 4-1: PQL/PH Architecture

4.1.3 The Two-pass Design

The main idea of a two-pass design is that PQL interprets an end-user query in two-passes. The first pass involves translating a high-level process query language into a lower level, intermediate form of query language that can more readily be used in a search algorithm or be translated into native database queries. Figure 4-2 shows a

conceptual PQL architecture using the two-pass design.

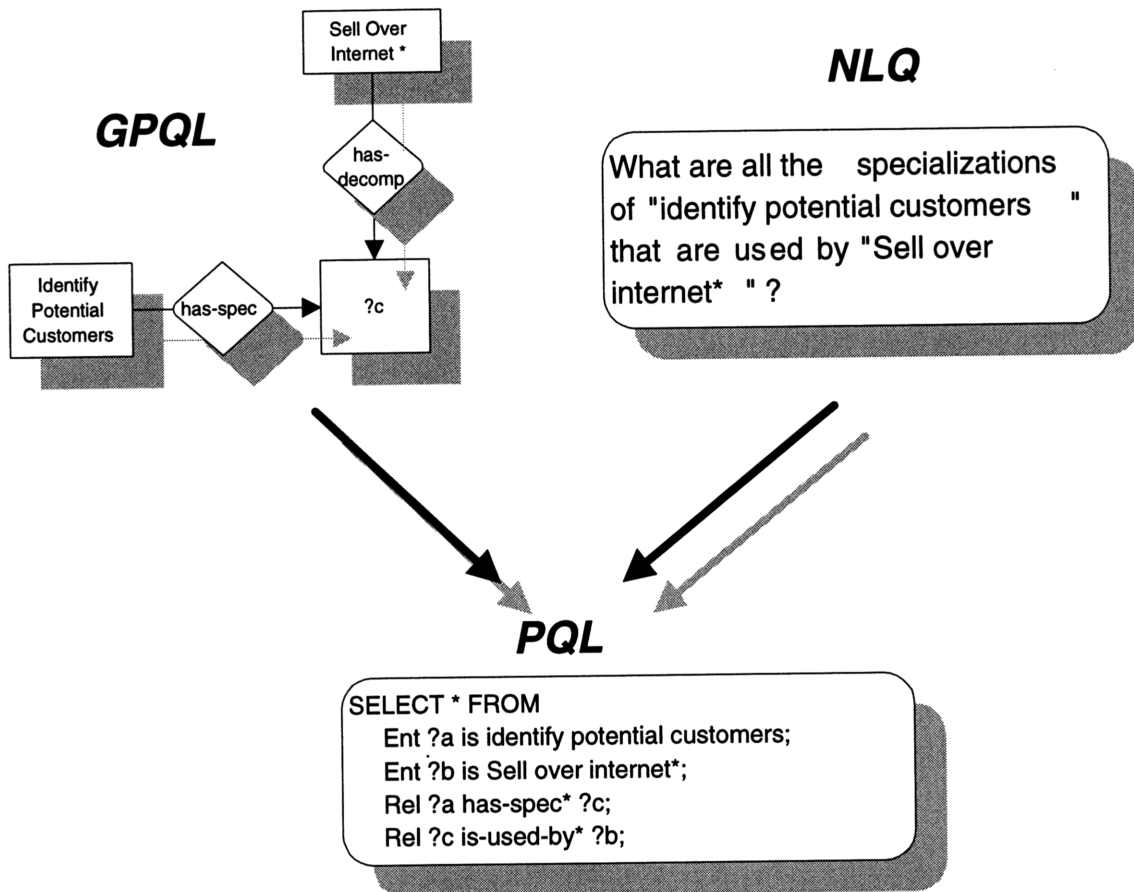


Figure 4-2: Two-pass design of PQL

The advantage of such two-pass architecture is that it allows us to tackle part of the problem while leaving the other parts flexible to the future (those parts that call for design decisions that we are not ready to make yet). One example of a design decision that we are not ready to make, as mentioned before, is the end-user interface of the query language. As figure 4-2 indicated, different end-user interfaces can be built on top of PQL. One powerful end-user interface that we are planning to build is GPQL (Graphical Process Query Language). GPQL allows the user to express the query in terms of the structure of process relationships. With drag and drop capability, GPQL lets the user build structures that express the relationships among different processes that can be either bound or unbound (processes that are

left unspecified or partly specified). Using the GPQL example in figure 4-2 as an example, we have a bounded process called *Identify Potential Customers*, a partially bounded process called *Sell Over Internet** [3] and an unbounded process which we named with the variable *?c*. The links between processes specify the relationships between them. *?c* is a specialization of *Identify Potential Customers* and is also used by *Sell Over Internet**. The answer to the query would be the set of all possible bindings to the unbound processes such that they satisfy the the relationships and constraints expressed in the graphical structure. Such type of approach is similar to the "Query by Example" type of approach.[5]

We are not limited to only one particular user interface as figure 4-2 shows. The other user interface, called NLQ (Natural Language Query), provides a syntax that resembles natural language that is restricted for expressing process queries. NLQ employs a higher level language syntax: one that is very close to what coordination science researchers use in their everyday discussions. For example, for the same query that was expressed graphically in GPQL, a coordination scientist would express it as, "What are all the specializations of Identify Potential Customers that are used by *Sell over Internet** ?"

There will surely be new ideas or needs for other kinds of interfaces to PQL in the future and the two-pass design has the flexibility to accommodate such additions. It is easy to imagine that users would soon demand an interface that can be used as part of PH_WIN where the user can drag and drop entities which are graphically displayed in one of the viewers into a query construction window. All of these additions to PQL are straightforward provided that it is possible to write an interpreter which translates the higher level queries into PQL form. This condition demands that PQL be able to express any kind of complex relationships among processes and any types of queries that the user would be interested in making. Here we trade convenience for completeness. We use an entity-relation model for PQL because we think that an entity-relation model can express any combinations of relationships, however complex (claimed here without proof). The examples of GPQL and NLQ would both translate

[3] Partially bounded because of the wild card string matching operator.

to the following PQL statements:

ENT ?a is Identify potential customers;
ENT ?b is Sell over internet *;
REL has-spec* ?c;
REL is-used-by* ?b;

The first statement binds the variable *?a* to *Identify Potential Customers* and the second statement binds the variable *?b* to processes that have the name *Sell Over Internet**. The third and fourth statements specify the relationship between the new unbounded variable *?c* and the bounded variables. One immediate advantage is that our GPQL can be translated straightforwardly into such entity-relation model because GPQL is really nothing else but a graphical description of an entity-relation structure.

Using such a two-pass approach allows us to divide and conquer. In particular, it allows us to build PQL first, concentrating on the search and match algorithms and exploring the feasibility and efficiency of different query methods. Research scientists would not feel uncomfortable using PQL through a very simple interface. Better and user friendlier interfaces can be postponed for later.

4.1.4 An Entity-Relation Approach

We use an E-R model for the syntax of PQL because an E-R model can express any possible sub-pattern of the Process Handbook database. The entity-relationship (E-R) model is designed to model a world that is consists of a set of basic objects called entities and relationships among these objects. An entity is an object that exists and is distinguishable from other objects. A relation is an association among several entities. Formally, if $E_1, E_2, \ldots, E_n$ are entity sets, then a relation set R is a subset of $\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$ where $(e_1, e_2, \ldots, e_n)$ is a relation. It can be proved that it is always possible to replace a non-binary relationship set by a number of distinct binary relationship sets. This means that we

can always model any entity-relation with only binary relationship sets. Attribute describes some entity's property that we do not consider an entity in its own right or we simply chosen not to model it as an entity.

Applying this perspective to the Process Handbook, we find that entities are activities, dependencies, ports, bundles and etc. Relations are "decompositions", "specializations", "is-used-by", "generalizations", and etc. Attributes are entity names, entity descriptions, and other Process Handbook attributes. For example, in figure 4-3, the activities *Identify Potential Customers, Sell Over Internet Mall* and *Attract Potential Customers to Web site* are entities in our E-R model. Entities are depicted in regular boxes. The diamond shaped "has-decomp" and "has-spec" are two binary relations connecting the appropriate activities. The arrows specify the direction of the relation if the relation is directional. As in the case of figure 4-3, the arrow indicates that *Attract Potential Customers to Web site* is a specialization of *Identify Potential Customers.* Or speaking in the same direction of the arrow, *Identify Potential Customers* has specialization *Attract Potential Customers to Web site.* Similarly, the arrow shows that *Sell Over Internet Mall* has *Attract Potential Customers to Web site* in its decomposition. The entity names such as *Attract Potential Customers to Web site*, *Identify Potential Customers*, and *Sell Over Internet Mall* are actually attributes of their activities. Since entity name is such a frequently used attribute of an entity,[4] we simplify our E-R diagrams by putting entity name attribute directly inside the entity rectangle. Generally, attributes are represented as ovals with a link to the entity that they modify in E-R diagrams. In the diagram, *Attract Potential Customers to Website* has an *Skills Required* attribute which has the value of *Website design.*

[4]Actually, entity ID is a better identifier for a process, because entity IDs are unique while entity names are not guaranteed to be unique. However, since we are human and not machines, using entity names makes more sense to us

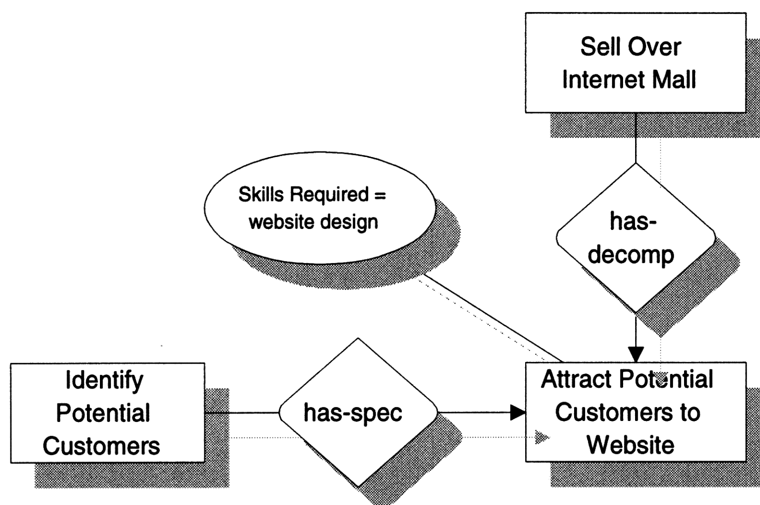


Figure 4-3: An Entity-Relation Diagram

4.1.5 Matching ER Graphs

Having descibed a Handbook process structures with the ER language, we now need a way to search the database for processes that match such relationships. We refer to this task as *matching an ER graph*, since we are practically searching for a set of processes that we can place into the same ER graph while satisfying all the relationship constraints.

We use a standard tree searching approach to matching a ER graph. Each statements in our PQL query is treated as a primitive query.[5] A primitive query is one that we find the answers using lower level methods such as SQL or general purpose language procedures that use control sequence and logic on top of SQL to process more complex queries. On the PQL level, these primitive queries simply return a list of matches. This list identifies new branches to explore. Those that satisfy the existing constraints are in turn expanded further. Those that do not, are pruned away. When a path of the tree is fully expanded, [6] the nodes along that path of the tree correspond to the set of processes that satisfies the PQL query.

Figure 4-4 illustrates an instance of a PQL search tree. From the *Start* node, PQL

[5]There are only three types of primitive queries, as will be explained in the PQL Syntax section.
[6]A path is fully expanded when it has exhausted all the queries.

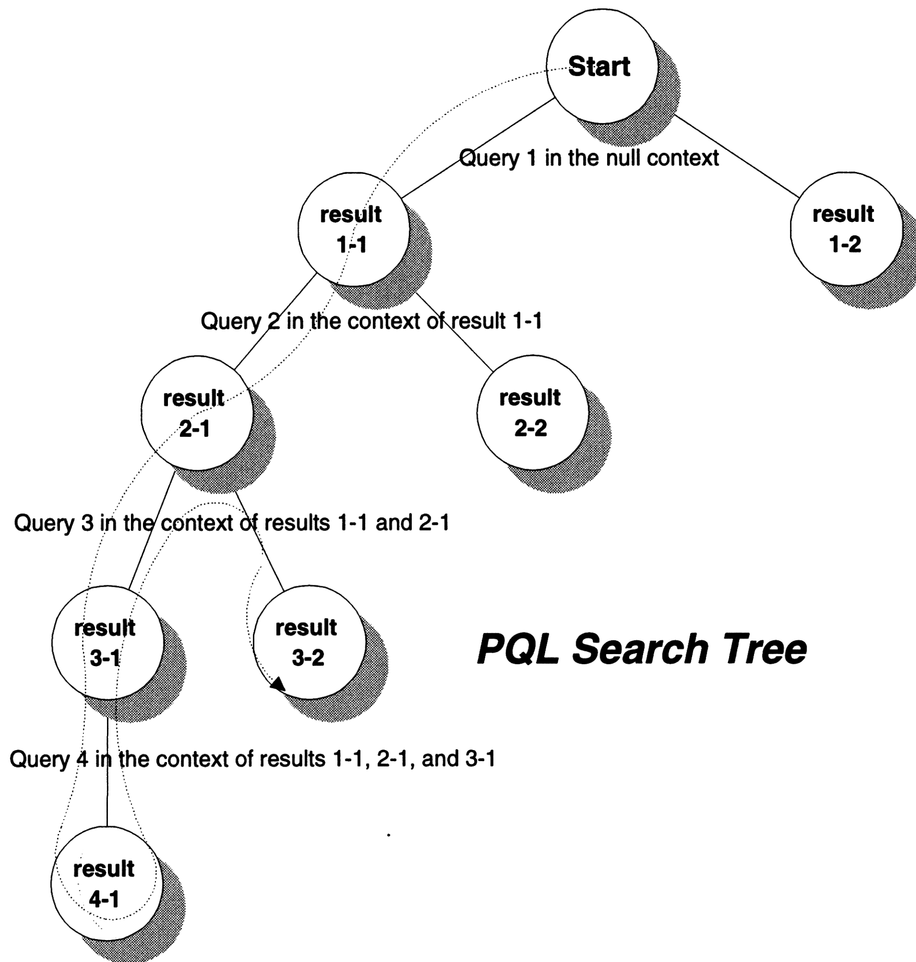


Figure 4-4: PQL Search Tree

processes the first primitive query and expands the tree with the results (*result 1-1*,[7] *result 1-2*, etc ...). Then, for each new node of the search tree, PQL processes the second query in the context of node that PQL is expanding. To make this important point clear, suppose that in figure 4-4 that *result 1-1* is $?a = A$ and *result 1-2* is $?a = B$. And further suppose that *query 2* is "REL ?a has-decomp ?b". When PQL expands node *result 1-1*, *query 2* is performed in the context of $?a = A$. The results will be a set of pairs (?a, ?b) such that $?a = A$ for all pairs and ?b is bounded to each sub-activities of A. When PQL is expanding the node *result 1-2*, the context is $?a = B$ and we are looking for subactivities of B in this case. Not all queries generate

[7] The notation of ***result a-b*** indicates that it is the b-th result from the a-th query.

new branches. If a query involves variables that are already bounded in the context, the result is simply an acceptance or rejection of the branch. In figure 4-4, *query 4* on *result 3-1* looked like such behavior; the branch is simply extended after evaluating the *query 4* to true in the context of *result 3-1*. When a branch reaches n level deep, where n is the number of primitive queries in the PQL query, then we have found a match. The last node of such a branch will contain a context that satisfies every primitive PQL queries, and therefore, the whole query.

We have experimented with different searching algorithms and found so far that a depth-first search is the most appropriate for our purposes. Readers who are not familiar with searching algorithms, or would like to refresh their memory, should refer to Appendix B. The reason is that all PQL search tree paths either reach dead ends or become complete paths after a constant number of steps. In fact, the length of a PQL search tree path is at most the number of primitive queries in the PQL query. Depth-first, therefore, will know whether a path is a success or a dead end in a short amount of work. PQL search trees are usually shallow and fat (branching factor) makes breadth-first search a bad approach because the size of the tree explodes exponentially. Depth-first search expands a local portion of the tree that it is working on and that portion can be deallocated when done with.

Depth-first search has one additional advantage; it can provide partial results as the search proceeds. This property allows for the option of terminating the search after a specified number of matches are found or the user is already satisfied with the results given so far. In figure 4-4, the dotted line traces the path that would be taken by a depth-first search.

4.2 PQL Syntax and Simple Queries

PQL is designed specifically for expressing entity-relation structures efficiently. Its has a few simple syntax and constructs. As mentioned in section 5.1.3, there are three types of objects in an entity-relational world: entities, relations, and attributes. Attributes are things that we consider not worthwhile to model as full entities and

relations specify how two entities relate to each other. Variables are used to hold entities or sets of entities. PQL variables must begin with a "?" as the first character of the variable name. This section describes the syntax of PQL constructs and present a few simple examples of their use.

4.2.1 Entity Operation

The ENTITY operation binds a variable to an entity or entities. ENTITY statements have the form:

ENTITY ⟨var⟩ is [⟨type⟩] ⟨name string⟩;

var := ?⟨string⟩

type := Activity | Port | Dependency | etc ...

The optional argument ⟨type⟩ specifies the type of the entity (e.g., activity, port, bundle, dependency, resource, etc...). If the type is not explicitly specified, the default is activity, which is also the only type of entity supported in the current implementation of PQL. Future extensions of PQL will allow the querying of ports and dependencies and at that time we can use the ENT[8] statement "ENT ?x is PORT Consumer of Coffee Beans" to bind ?x to a port.[9]

Entity name string is the name of the entity that will be bound to the variable ⟨var⟩. You can use "*" for wildcard string matching in the entity name. If more than one process has entity name matching the wildcard string then ⟨var⟩ is bound to the set of processes with matching names.

ENT ?x is Sell over Internet*;

((?x Sell over internet 3122))
((?x Sell over internet mall 3140))

Figure 4-5: Example of PQL Entity Query

[8] ENT is the abbreviated form of ENTITY. PQL recognizes both versions.

[9] We will need to distinguish between ¡type¿ specification and the first word of ¡name string¿.

The query in figure 4-5, for example, binds any activity (default) that have the string "Sell over Internet" in the beginning of their name. The result of the query shows that there are two such processes.

Query results are displayed in the form of list of bindings as illustrated by figure 4-6. Figure 4-6 shows a list of m matches where each match is a unique n-tuple of bindings. Each set of bindings, ($(?x_1^k\ name_1^k\ id_1^k)$ $(?x_2^k\ name_2^k\ id_2^k)$ $\ldots$ $(?x_n^k\ name_n^k\ id_n^k)$), where $1 \leq k \leq m$, shows the variable $?x_j$, $1 \leq j \leq n$, and the set values they are bound to that satisfy the query. A binding of variable $?x_j$ to a process with ID=id in the k-th match is shown as $(?x_j^k\ name_j^k\ id_j^k)$, where the $name_j^k$ is added to help us make sense of the results. If the results were to be interpreted by a higher level application, the pair $(?x_j^k\ id_j^k)$ is sufficient.

$$((?x_1^1\ name_1^1\ id_1^1)\ (?x_2^1\ name_2^1\ id_2^1)\ \ldots\ (?x_n^1\ name_n^1\ id_n^1))$$

$$((?x_1^2\ name_1^2\ id_1^2)\ (?x_2^2\ name_2^2\ id_2^2)\ \ldots\ (?x_n^2\ name_n^2\ id_n^2))$$

$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$$

$$((?x_1^m\ name_1^m\ id_1^m)\ (?x_2^m\ name_2^m\ id_2^m)\ \ldots\ (?x_n^m\ name_n^m\ id_n^m))$$

Figure 4-6: PQL Result Format

In our ENTITY example in figure 4-5, there are two matches but only one variable in each match. Therefore, $?x_1^1$ is bound to ?x, $name_1^1$ to "Sell over internet", and id_1^1 to 3122 for the first match. The second match has $?x_1^2$ bound to ?x, $name_1^2$ to "Sell over internet mall", and id_1^2 to 3140.

4.2.2 Attribute Operation

The ATTRIBUTE operation allows the user to express queries based on the value of entity properties. ATTRIBUTE statements have the form:

ATTRIBUTE ⟨var⟩ ⟨attribute name⟩ ⟨op⟩ ⟨value⟩;

var := ?⟨string⟩

attribute name := Name | Description | Contact

op := $< \mid = \mid > \mid \leq \mid \geq \mid \neq$

Currently, only the attributes "entity name", "entity description", and "entity contact" are implemented and only the operator "=" can be used for these attributes. Attribute values is the value that you are trying to match and you can use "*" for wildcard string matching. Notice that no quotes are needed for string values in PQL.

When two or more ATT[10] statements are used together, the results are those processes that satisfy all ATTRIB statements simultaneously (Statements in a query are implicitly joined by AND operation). For example, the query in figure 4-7 finds processes that have both "DARPA" and "AIMS" in their descriptions. [11]

```
ATT ?a desc = *darpa*;
ATT ?a desc = *aims*;

( (?a Find source - pre-certified agile suppliers 4977) )
( (?a Agile manufacturing 5810) )
( (?a New as of August 6200) )
( (?a Search electronic catalog 6246) )
( (?a Select supplier using agents 6249) )
( (?a Negotiate contracts using information technology 6251) )
```

Figure 4-7: Example of PQL Attribute Query

4.2.3 Relation Operation

Both the ENTITY and ATTRIBUTE operations describe queries about entities. A RELATION operation provides a way to describe how one entity is related to another process. RELATION statements have the form:

RELATION ⟨var1⟩ ⟨relation type⟩ ⟨var2⟩;

var1, var2 := ?⟨string⟩

relation type := has-decomp | has-spec | is-used-by | has-gen

[10] ATTRIBUTE can be abbreviated as ATT in PQL queries.

[11] Description can be abbreviated to desc in PQL queries.

The two variables of the REL[12] statement are entities. The four relation types are listed here along with their meaning:

1. has-decomp ⟶ has-decomposition
2. is-used-by ⟶ has-composition
3. has-spec ⟶ has-specialization
4. has-gen ⟶ has-generalization

PQL currently supports these four basic Process Handbook relationships. Figure 4-8 illustrates the use of RELATION statement.

```
ENT ?a is Sell over Internet;
REL ?a has-decomp ?b;

( (?a Sell over internet 3122) (?b Obtain order 2804) )
( (?a Sell over internet 3122) (?b Receive payments 2387) )
( (?a Sell over internet 3122) (?b Identify potential customers' needs 3281) )
( (?a Sell over internet 3122) (?b Identify by customer self-identifying 3349) )
( (?a Sell over internet 3122) (?b Advertise via internet 4948) )
( (?a Sell over internet 3122) (?b Deliver the thing 2623) )
```

Figure 4-8: Example of PQL Decomposition Query

The query returns the subactivities of *Sell over Internet.* Variable *?a* is bound to *Sell over Internet* by the first statement and the second statement specifies that *?b* is a decomposition of *?a.* The set of tuples (pair in this case) that satisfies this query is the set where *?a* is bound to *Sell over Internet* and *?b* is bound to each of the subactivities of *?a.* For another example, consider another, but similar, query that gets the specializations of a process as shown in figure 4-9.

RELATION statements must use variables. Entity names can not substitute for variables inline; names must be bound in a separate ENTITY statement. The reason is that we want to have simple and uniform length PQL primitive statements.

[12]RELATION can be abbreviated with REL in PQL queries.

```
ENT ?a is Sell something;
REL ?a has-spec ?b;

( (?a Sell something 2639) (?b Sell in retail store 3120) )
( (?a Sell something 2639) (?b Sell electronically 4935) )
( (?a Sell something 2639) (?b Sell using distribution channels 5574) )
( (?a Sell something 2639) (?b Sell by direct marketing 3121) )
( (?a Sell something 2639) (?b Sell resources 6077) )
( (?a Sell something 2639) (?b Sell outputs 6078) )
( (?a Sell something 2639) (?b Market and sell 3476) )
( (?a Sell something 2639) (?b Sell products and services 3484) )
( (?a Sell something 2639) (?b Sell products 5294) )
( (?a Sell something 2639) (?b Marketing- Kotler 6155) )
( (?a Sell something 2639) (?b Sell something to DARPA 5705) )
( (?a Sell something 2639) (?b Deliver products - SCOR 6397) )
```

Figure 4-9: Example of PQL Specialization Query

Both of the decomposition and specialization queries above are immediate operations. That is, the processes returned are all immediate decompositions (specializations) of process ?a. To get *all* decompositions (specializations) of *?a*[13], one must use the recursive operators *has-decomp** (*has-spec**). The user should keep in mind that the levels of decomposition (specialization) are arbitrary and dependent on the particular way the user decides to model the processes. Therefore, decomposition generally mean *all* decompositions and recursive operators should be used.

You can tell PQL specifically how many levels of decomposition (specialization) to return by appending "*" with an integer such as has-decomp*3 tells PQL to go up to three levels deep in the decomposition hierarchy. This option is meant to be used mainly by users who have special knowledge of the process database or for users who are concerned with the details of the process handbook hierarchies. This option can also be used to limit the amount of results returned, especially when you are not interested in the very specialized processes.

[13]All decompositions of ?a means the immediate decompositions of ?a and the decompositions of the immediate decompositions of ?a and so on down the decomposition hierarchy.

Is-used-by and *has-gen* are the inverses of *has-decomp* and *has-spec* respectively. Specifically,

?a has-decomp ?b $\longleftrightarrow$?b is-used-by ?a

?c has-spec ?d $\longleftrightarrow$?d has-gen ?c

Theoretically, having has-decomp and has-spec is sufficient. However, Is-used-by and has-gen provide an alternative to expressing a relation in a more straightforward and natural way.

4.2.4 Select Operation

The SELECT operation is the counterpart of the PROJECTION operation in relational algebra.[14] SELECT ... FROM serves to project only the variables that you are interested in seeing from the results of the enclosed queries. The SELECT statement encloses the list of queries and it has the form,

SELECT $?x_1$, $?x_2$, ..., $?x_n$ FROM
    ($query_1$; $query_2$; ...; $query_m$;)

where $?x_1$, $?x_2$, ..., $?x_n$ are the variables to be projected. If the wildcard operator "*" is used, then SELECT projects everything.

Consider the "decomposition of *Sell over Internet*" query. It is not really that interesting to see ?a bound to *Sell over Internet* in the results because we know what ?a is; it is the decompositions of ?a that we want to find. Therefore, it makes sense to SELECT only ?b for the result of our query as figure 4-10 shows.

4.2.5 Logical Operation

Logical operators are used to combine different sets of results together. The two operators currently implemented are AND and OR. Since our PQL graph matching algorithm searches for a set of bindings that satisfies all primitive PQL queries, it has the same effect as having many primitive queries as full queries and joining the

[14]This serves the same function as the SQL SELECT operation.

```
SELECT ?b FROM
    ENT ?a is Sell over Internet;
    REL ?a has-decomp ?b;

( (?b Obtain order 2804) )
( (?b Receive payments 2387) )
( (?b Identify potential customers' needs 3281) )
( (?b Identify by customer self-identifying 3349) )
( (?b Advertise via internet 4948) )
( (?b Deliver the thing 2623) )
```

Figure 4-10: Example of PQL Query with SELECT...FROM

results by the AND operator. However, the latter approach is much less efficient than the former, because the knowledge of previous queries is not used to limit the possibilities of the current query.[15] Figure 4-11 illustrates the use of AND on two primitive attribute queries. This query is identical to the the query in figure 4-7; therefore, the results will be identical as well. In general, the AND operation is the Cartesian product of the two lists. In the degenerate case when the two lists have the same variables, AND operation is just the intersection of the two lists.

```
(AND
   (ATT ?a desc = *darpa*;)
   (ATT ?a desc = *aims*;) )

( (?a Find source - pre-certified agile suppliers 4977) )
( (?a Agile manufacturing 5810) )
( (?a New as of August 6200) )
( (?a Search electronic catalog 6246) )
( (?a Select supplier using agents 6249) )
( (?a Negotiate contracts using information technology 6251) )
```

Figure 4-11: Example of PQL Query with Logical Operator

OR operation is simply the union of the two binding lists. Unlike AND, OR

[15] This tells us that any PQL query with AND operation can be re-written without the AND and the resulting query would be more optimized.

is a necessary operator, in the sense that it cannot be rewritten with other PQL constructs.

The NOT operator is not implemented in this version of PQL because we have yet to decide on a workable specification of the operation. In general, if ?a is a set of processes of size n, then NOT ?a is a set of size $\mathcal{U} - n$, where $\mathcal{U}$ is the universe of our Process Handbook database. In most cases, $\mathcal{U} - n$ is practically infinite. Therefore, it is not practical to perform a NOT operation alone; it generates too much data to return. However, it is perfectly practical to have ?a AND NOT ?b where the size of the result is well bounded. Therefore, we need a method to delay the execution of NOT until it can be executed with another operation such as AND or OR. We need to do more research on how to provide such "lazy" execution scheme before we feel comfortable in allowing the usage of the NOT operator in PQL.

4.3 Complex Queries

You have already seen from section 4.2 what primitive queries can do. However, the power of PQL comes from complex queries, the combining of two or more primitive queries to describe more interesting constraints. A useful category of complex queries is called "Cross-sectional" queries. The name "Cross-section" comes from the queries' property of spanning two or more dimensions. All primitive queries span only one dimension while cross-sectional queries search for the intersections of two or more primitive queries spanning different dimensions. A very popular type of cross-sectional query is that of having two instances spanning the same dimension linked by some property that spans the second dimension. The two dimensions are the two dimensions of the coordination compass: decomposition and specialization. The query usually have the form, "What are all the specializations $\mathcal{X}$ that are used by some specialization of $\mathcal{Y}$."

Recall in chapter 1 that we posed a couple of interesting queries arising from different professional scenarios. In this section, we will show how to construct PQL versions of those queries. We will also see the kind of results that are coming back

from those queries.

4.3.1 The Internet Consultant's Query

We first posed the Internet consultant's problem in section 1.2.1. There are two parts to the problem. The first is the question, "What are the different ways of selling something that involves the Internet and is more cost effective than its non-Internet counterparts?" This is a qualitative query.[16] We can't answer this question with our current PQL technology, but we should be able to in future versions of PQL. Here, we will answer a simpler version of the above, namely, "What are the different ways of selling something that involves the Internet?" In coordination science jargon, this query is correctly expressed as "What are all the specializations of *Sell Something* that have Internet in their descriptions?" Figure 4-12 illustrates how this query is expressed in PQL and the results that satisfied the query.

```
SELECT ?b FROM
    (ENT ?a is Sell something;
    REL ?a has-spec* ?b;
    ATT ?b desc = *internet*;)

( (?b Sell electronically 4935) )
( (?b Sell over internet 3122) )
( (?b Consortium for Purchasing and Distribution 5018) )
( (?b Oasis 5022) )
( (?b Sell through on-line mall 4961) )
( (?b CUC International Shoppers Advantage 5023) )
( (?b Toyota USA 5024) )
( (?b Blockbuster Video 5025) )
( (?b Sell over internet mall 3140) )
( (?b DealerNet 5026) )
( (?b CUC International : Auto Vantage 5027) )
```

Figure 4-12: Query: "What are the different ways of selling something that involves the Internet?"

[16] Qualitative and quantitative queries are discussed in chapter 6, Future Work. These queries involved the comparisons of processes with respect to some attributes.

In figure 4-12, ?b is bound to all the processes that are specializations of *Sell Something* and involve the Internet in some ways. The results ranges from high level process *Sell electronically* to specific process *Sell Through On-line Mall* and *DealerNet*. It may not be clear how some result processes involve the Internet. Nevertheless, they all have "Internet" in their process descriptions.

Having found the sales processes involving the Internet, the next step is to figure out how to identify potential customers. This leads us to the second part of our Internet Consultant's problem, "What are the different ways to identify potential customers if you want to sell over the Internet?" Again, the question re-phrased in coordination science language is, "What are all the specializations of *identify Potential Customers* that are used by *Sell Over Internet*?" Figure 4-13 shows the PQL realization of this query.

```
SELECT ?b, ?c FROM
    ENT ?a is Identify potential customers;
    ENT ?b is Sell over internet*;
    REL ?a has-spec* ?c;
    REL ?c is-used-by* ?b;

( (?b Sell over internet 3122) (?c Identify by customer self-identifying 3349) )
( (?b Sell over internet mall 3140) (?c Attract potential customers to website 6074) )
```

Figure 4-13: Query: "What are the different ways to identify potential customers if you want to sell over the Internet?"

It is worthwhile to trace through this query and explain where the answers are coming from. First notice the wildcard matching in the name *Sell Over Internet**, which matched two processes, *Sell over Internet* and *Sell over Internet Mall*.[17] Figure 4-14 shows a snapshot of part of *Identify Potential Customer*'s specialization tree.

Notice where *Identify by Customer Self-identifying* and *Attract Potential Customers to Website* are in the specialization tree. These processes are used by *Sell over Internet* and *Sell over Internet Mall* respectively as shown in figure 4-15 and

[17] *Sell over Internet Mall* is actually a specialization of *Sell over Internet*.

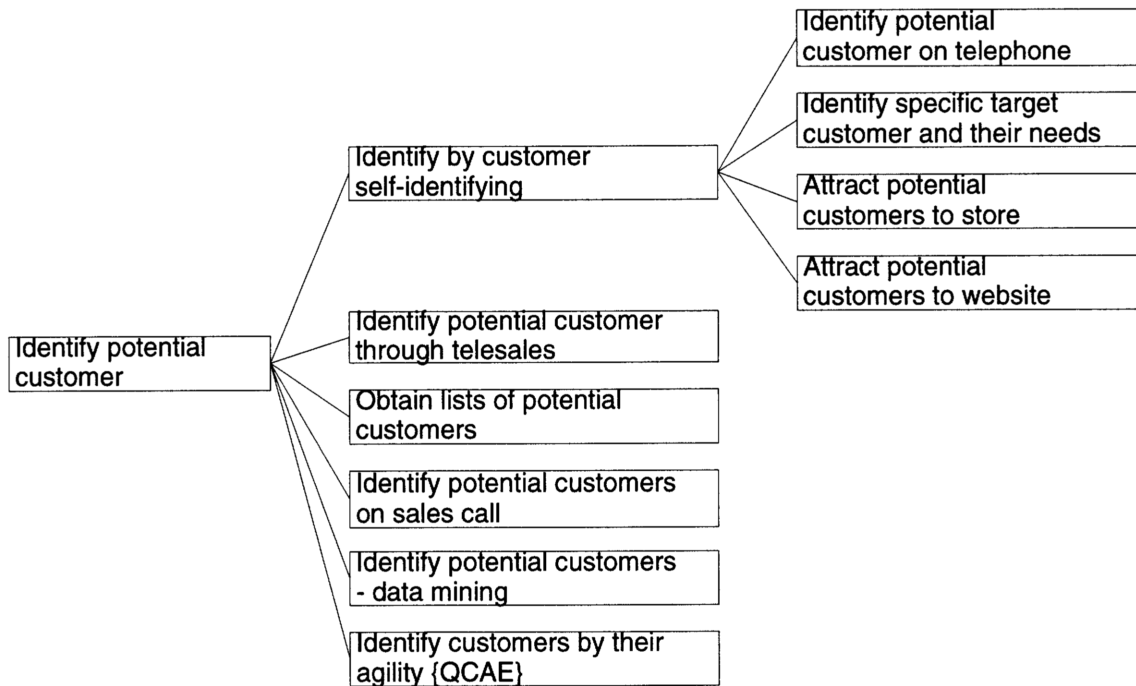


Figure 4-14: Specialization of *Identify Potential Customer*

figure 4-16.

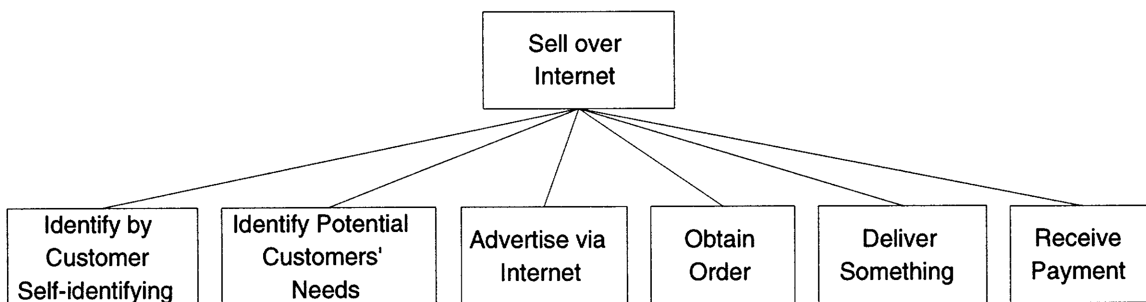


Figure 4-15: Decomposition of *Sell Over Internet*

There are two matches for the query. The first pair shows that *Sell over Internet* uses *Identify by Customer Self-identifying* as the way of *Identify Potential Customer.* Since this pair of processes are relatively high-level, it may be a bit abstract to the user. The answer basically tells us that in general, the way to identify potential customers on the Internet is by letting the customers identify themselves. The second

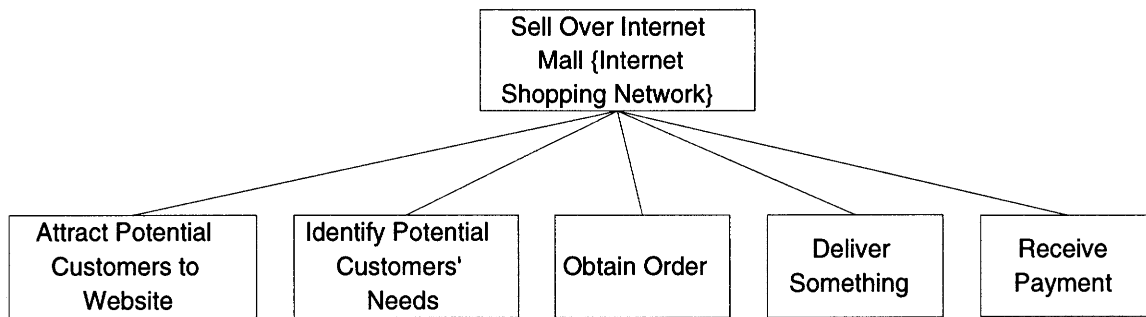


Figure 4-16: Decomposition of *Sell Over Internet Mall*

pair will make the idea more concrete since it is actually a specialization of the first pair. *Sell over Internet Mall*, a more specific kind of *Sell over Internet* uses *Attract Potential Customers to Website* as a special way of *Identify by Customer Self-identifying.* On the Internet, the way to identify potential customer is to attract people to your website and through this process, you have identified your potential customers. The reason is that those who came to your website came because they are interested in things that are related to your website. In effect, they have self-identified as your potential customers.

"ENT ?b is Sell over internet*" is a bad way to express processes that are sell over internet "anything"; it is exactly what we criticize keyword search for. String matching may generate false positives as well as false negatives. The correct way of expressing all kinds of *Sell Over Internet* processes is by getting all of its specializations as the following query shows.

```
SELECT ?b, ?c FROM
    ENT ?a is identify potential customers;
    ENT ?b is Sell over internet;
    REL ?a has-spec* ?c;
    REL ?b has-spec* ?d;
    REL ?c is-used-by* ?d;
```

This query spans two instance of the specialization dimension and is linked by the decomposition dimension. We will see such type of query again later. We show the simpler query for *Sell over Internet* here to demonstrate the wild card binding of ENT

statements .

4.3.2 Operations Manager's Query

Our second problem is the operations manager's query. Here, the question is, "What are all the sales processes that involve dealing with DARPA?" In coordination science jargon, the question is "What are all the specializations of *Sell Something* that have DARPA in their description?" This query has exactly the same structure as our first Internet query. Having traced through a query in the previous section, we will work through the next two examples quickly. Figure 4-17 shows the realization of the query in PQL and the resulting matches from the Process Handbook database.

```
SELECT ?a, ?c FROM
    (ATT ?a desc = *DARPA*;
    ENT ?b is Sell something;
    REL ?b has-spec* ?c;
    REL ?a is-used-by* ?c;)

( (?a Identify customer needs 4989)
  (?c Sell something to DARPA 5705) )
( (?a Inform potential customers over Internet 4990)
  (?c Sell something to DARPA 5705) )
( (?a Calculate cost using Activity Based Costing 4993)
  (?c Sell something to DARPA 5705) )
```

Figure 4-17: Query: "What are all the sales processes that involve dealing with DARPA?"

4.3.3 Human Resources's Query

Our last problem is the human resources's query, "What are all the ways of identifying sources that are used by a type of hire human resources?" Stated in coordination science terms, the question is, "What are all the specializations of *Identify Source - How?* that are used by some kind of *Hire Human Resources* ?" Figure 4-18 shows the PQL query and results.

```
SELECT ?c, ?d FROM
        (ENT ?a is Hire human resources;
        ENT ?b is Identify source - how?;
        REL ?a has-spec* ?c;
        REL ?b has-spec* ?d;
        REL ?d is-used-by* ?c;)

( (?c asdl fh 10292)
  (?d Find source (purchasing) 2808) )
( (?c Candidate driven senior hire 5510)
  (?d Find source by self-identification 5417) )
( (?c Fully standardized senior hire 5511)
  (?d Find source by search firm 5442) )
( (?c Hire senior position with sourcing by journal 5743)
  (?d Find source via publications 5718) )
( (?c Hire senior position with sourcing by search firm 5744)
  (?d Find source by search firm 5442) )
( (?c Hire senior position using networks 5745)
  (?d Find source by networking 5443) )
( (?c Hire senior position with self identification 5746)
  (?d Find source by self-identification 5417) )
( (?c Source senior person via search firm 5533)
  (?d Find source by search firm 5442) )
( (?c Source senior person via networking 5534)
  (?d Find source by networking 5443) )
```

Figure 4-18: Query: "What are all the ways of identifying sources that is used by a type of hire human resources?"

The results show pairs of specific *Identify Source - How?* that is directly used by a specific kind of *Hire Human Resources* ?" Such query would be very time consuming if not inconceivable without PQL.

Chapter 5

Evaluation

In the beginning of chapter 4, we have listed the basic design goals of PQL. In this chapter, PQL will be evaluated with respect to those design goals. I will re-iterate the design goals here.

1. High level, concise, intuitive and easy to learn language
2. Database independent
3. Increased precision (reduce false positives and false negatives)
4. Straightforward UI extension
5. Optimizable

5.1 High Level

PQL is high level, concise and intuitive. The simple and mimimal set of syntax makes it easy to learn for users, even though it is intended that users' interaction with PQL will ultimately be through higher level user-interfaces. The PQL syntax, modeled after entity-relation style, is quite similar to the language of the Process Handbook, which makes query expression intuitive and direct.

5.2 Database Independent

PQL is database independent. The PQL syntax is completely based on a entity-relation model which is independent of any database representation. Furthermore, the implementation of PQL is mostly database independent; the PQL tree search engine, binding/set/list objects, parser, and even most of the primitive query modules are completely database independent. PQL encapsulates database accessing functions into a few methods that abstracts the database away nicely (i.e., *getDecomp*, *getSpec*, etc ...). PQL can be easily ported to different databases or tolerate any database model changes.

5.3 Increased Precision

PQL increases precision in terms of reducing false positives and false negatives. Comparing PQL results with those from keyword searches, we have made much progress in precision. The added precision comes from semantic (structure-based) rather than purely substring search. Furthermore, the capability of PQL to return multi-dimensional results (a list of binding sets where each set is an instance of one or more variables binding to values that satisfy the query) is very powerful. Chapter 6 suggests the idea of in-exact matching (matching by semantic closeness) which can further increase query precision.

5.4 Potential UI Capability

PQL currently has a very simple user interface, mainly for the purpose of experimentation. The current PQL UI has a query input window, a output window for results, and a message window for system messages and other statistics. However, better and more user friendly UIs can be implemented on top of PQL easily. Chapter 4 already suggested two possible high level UIs: GPQL (Graphical Process Query Language) and NLQ (Natural Language Query). Implementing high level UIs, involves translating the high level UI metaphors into PQL syntax. In the PQL syntax design

process, we strived for the simplest and the minimal set of instructions necessary for the language.[1] Therefore, translation from higher level UIs is simple and direct.

5.5 Optimizable

Because of our design decision to implement PQL at the API level, PQL is highly optimizable. Not much attention was paid to optimization at such early stage of project PQL. Nevertheless, PQL is designed to be highly optimizable. Because of its modular design, many PQL components can be replaced easily. For example, more efficient search algorithms can be plugged and played without affecting other parts of PQL. More optimized versions of binding sets and list objects can be substituted for current implementations. Chapter 6 suggests query re-ordering and heuristic search as future optimizations for PQL.

[1]Our approach here is similar in philsophy to the RISC approach in computer architecture and many of the advantages can be inferred from the analogy.

Chapter 6

Future Work

This thesis work is only the start of some interesting and important research. The results of this experimental PQL project suggest that it is worthwhile to continue research in the area. Advances in PQL will change the way people use the Process Handbook. This section describes our visions of PQL: the important features that are left out of this research due to time constraints. Further research in PQL can be categorized into three areas. The first is in the design of high level interfaces for the end-user. The second involves the expansion of the language so more powerful queries can be made. The last is in the area of optimization: how to process queries efficiently to reduce search time as well as the kind of matches it returns.

6.1 Implementation of High Level Interfaces

High level interfaces are important because the end-users will ultimately be interacting with the Process Handbook through these. These interfaces should be tailored specifically to the purpose of the application. There are many creative ideas for high level interfaces as there are many different ways and purposes for querying the Process Handbook. We describe two examples of such interfaces here.

6.1.1 Graphical Process Query Language (GPQL)

GPQL helps the user in visualizing the structure and relationships of processes. Since users are already familiar with the visual metaphors of the Process Handbook viewers, we should adopt the same visual metaphors for GPQL. GPQL would be a good complement for PH_WIN and the two applications can greatly benefit from one another if integrated. With implicit queries (e.g., clicking on a process already displaying in one of the viewers and selecting "Find similar processes" from the pop-up menu), the user can save much time and effort. Look back to figure 4-2 to see how a graphical description of a process query can be directly translated into PQL.

6.1.2 Natural Language Query (NLQ)

NLQ is a high-level natural language text based interface. The semantics of NLQ should be intuitive and high-level, resembling the jargon of the coordination science community. There are many uses for NLQ. The query "What are all the specializations of *Identify Potential Customers* that are used by *Sell over Internet* ?" is easily understandable, even to users who are unfamiliar with the visual metaphors. Moreover, it is easily written down, recorded, or sent through email. NLQ may be the only way of querying the database when more interactive methods are not convenient such as PH_WEB.

6.2 Language Extensions

Another way of improving PQL is to extend the space of the language. Currently, PQL can only express queries about process entities and their relationships. As you have already seen in chapter 2, our process model is much more powerful and descriptive than that. It includes, for example, ports, dependencies, coordination mechanisms and trade-off matrices.

6.2.1 Expressing Dependencies in Queries

Adding the ability to express dependencies and the managing of dependencies would add another level of power to PQL. With such addition to the language, users can query processes with or without certain dependencies (e.g., "Show me all processes ?a and ?b that have dependency D between them.") For a process re-engineerer, the question "Show me all processes that can manage the dependency D." is crucial.

6.2.2 Qualitative and Quantitative Queries

Recall from section 1.2.1 that our Internet consultant is interested in the question, "what are the different ways of selling something that involves the Internet and is more cost effective than its non-Internet counterparts?" This is one example of qualitative queries. In addition to just expressing process structures, these queries can express qualitative comparisons among processes that share a common comparison dimension. Qualitative query *is* is important in process re-engineering, where the ojectives are usually to find alternative processes that accomplish the same goal but are more "cost-effective", "faster", or provide "better quality of service". Needless to say, being able to express qualitative queries is an important entension to PQL. Currently, qualitative comparison information is already present in the process database; it is modeled as bundles and tradeoff matrices. The challenge is to devise a systematic way of interpreting attributes and values of tradeoff matrices.

Quantitative queries go even further in describing attributes of processes. Consider the queries, "Find all the processes that can convert petroleum into plastic at $\mathcal{X}$ barrels/day." and "Find all hiring processes (specializations) with flow times below $\mathcal{X}$ days." These queries involve quantitative attributes of processes. Quantitative queries are useful to express more specific constraints in processes. Quantitative queries require an existing system of quantifying certain standard attributes of processes (e.g., time, cost). Users should be able to expect the existence of certain standard attributes from certain categories of processes while they should also be able to dynamically find out what attributes are specific to this process. Process

Handbook must also provide a way to specify the units used for the quantitative attributes and methods for conversion in case of comparison between mismatched units. Implementing quantitative queries is a challenging effort.

6.3 Optimizations

Optimization is very important to any query language. How users use the querying tool (if they use it at all) depends largely on the speed of the tool. This thesis have only put minimum emphasis on optimization such as the decision to use a depth-first search over a breath-first search algorithm. Many other potential optimizations come to mind, some are readily implementable while others require more research.

6.3.1 Query Re-ordering

A simple optimization that we have considered is query re-ordering. Queries are processed sequentially. Therefore, the result of a earlier query would have a major impact on the size of possibilities that later queries would have to consider. Logically, we would want to restrict the size of the bindings as early as possible, cutting down the possibilities to search later. Therefore, a good strategy is to process the queries that are more specific (expected to return relatively few matches) before we process those that are more general (expected to return many matches). Currently, this optimization is actually done by hand; the queries are entered in an optimal order according to the user's a priori knowledge of the process database. This is done effortlessly because the current users of PQL are all "super" users. Ultimately, we would need to rearrange the order of queries submitted by more naive users.

Query re-ordering is easier said than done. The number of matches for a given query is hard to predict in general. On a first pass, we would expect that LET statements are the most specific, serving to bind a variable to a single specific process most of the time. One might consider the next specific query type to be ATTRIBUTE statements. The last comes RELATION statements. Because RELATION statements involve two variables instead of one, one would want to delay the processing of such

query as late as possible or until we can bind as much of the two free variables as possible.[1] However, things are not always so straightforward. A LET statement, containing an unrestrained (or loosely-restrained) wildcard, would easily yield more matches any other query. The same can be said about popular attribute values in ATTRIBUTE statements. Therefore, our simple first-pass strategy does not always yield optimal ordering. A conceivable strategy may even do probing queries to get a better idea on how to order the queries.[2]

6.3.2 Heuristic Search

Search efficiency may be improved if we can define heuristics on our search. Heuristics are a measurement (approximation) that tells you how promising a choice looks in leading to an answer. We can use heuristics to order the choices so that the most promising are explored earliest. [12] We haven't made much effort to identify such heuristics yet, but the idea is worth some further research. The question is whether we can define any good heuristic from some a priori knowledge of the general structure of the Process Handbook database.

6.3.3 Inexact Matches

When we can't find any set of processes that matches our constraints exactly, we can return close matches or inexact matches, ordered for example, by their semantic closeness to the structure described in our query. Having the ability to return inexact matches, encourages users to specify more specific queries without worrying about not finding any matches and be forced to generalize the query in subsequent trials. Ordering the matches by semantic closeness helps the user focus on the relevant

[1] Theoretically, the statement, "RELATION ?x ⟨relation type⟩ ?y", with both ?x and ?y unbound, means all such relations in the database. In practice, especially when the database is significant in size, it is not practical to return all matches of this query. The current implementation of PQL makes this query illegal. However, given a query from a naive user, we have to be sure that the unbound RELATION statement is really unbound even after considering all the other queries. This makes query re-ordering a necessity, not just an optimization.

[2] Primitive queries are usually orders of magnitude faster than the tree search. Therefore, it may be worthwhile to do probing queries.

matches first.

Appendix A

The Process Handbook Architecture

The CCS Process Handbook project is applying the process modeling concepts discussed in chapter 2. The goal of the Process Handbook has been discussed previously, but in the process of developing this tool, we gained significant experience in analyzing processes in a formal way (e.g., decomposition, specialization, and dependency management). It also gave us a chance to experiment with the usability of our process model, to learn more about process modeling and to improve on the original model. Over the years, our process repository has grown significantly in size. The release of the web process handbook has increased the user population both in size and in geography.

A.1 Database Schema

Currently, the Process Handbook architecture stores all the data about processes in relational tables. The database schema divides up the information into seven relational tables: Entity, Attributes, Relation, Specialization, Relation_Types, Entity_Types, and OLEObjects. Figure A-1 shows the seven relational tables of the database schema.

It is important to realize one big difference between chapter 2, "The Modeling

of Organizational Processes" and "The Process Handbook Architecture" being described here. In the former, most of the design decisions made are for the correctness and expressiveness of the process model. In the latter, most of the design decisions made are for purpose of implementation and efficiencies. For example, specialization is one kind of relation in our process model but specializations are not included in the Relation table. The reason is that specialization is one of the most frequently used functions in the Process Handbook. Therefore, Specializations relations have its own table for efficiency in fetching the specializations of an entity. The Specializations table is implemented in such a way that it is very efficient to get the specializations of an entity (spec_child_ent_id). The straightforward way would be to include specializations in the Relation table and tag relation type (rel_type) to be spec. But this way involves a search of all relations that has a given starting entity ID (rel_start_ent_id) and is typed spec.

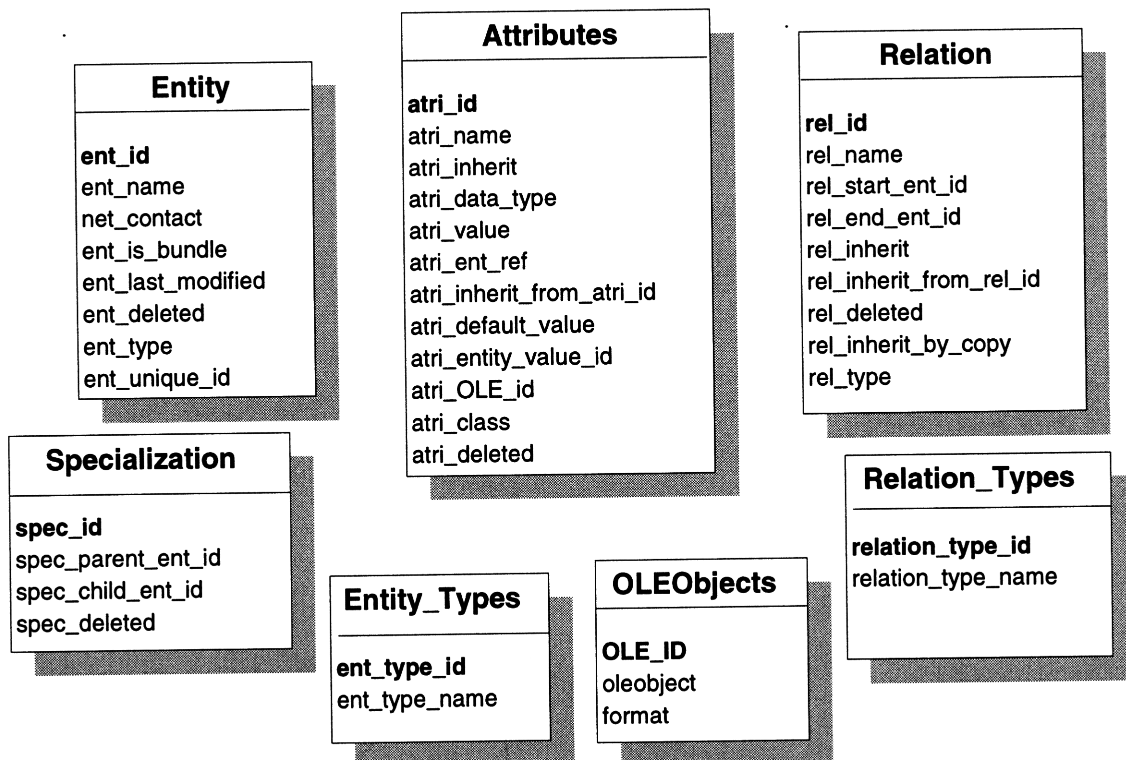


Figure A-1: The tables of the Process Handbook Database Schema

Another efficiency strategy that needs some explaining is the locations of attributes. Not all attributes are stored in the Attributes table; in fact, only the least used ones are stored there. The most commonly needed attributes for an entity (e.g., name, contact, last modified) are stored in the Entity table. Such implementation allows the most commonly needed attributes to be fetched along with the entity ID by just touching only one database table. Still, other frequently used attributes (e.g., managing entity ID for a dependency) are stored in the Relation table. These attributes will take longer to access than the essential ones in the Entity table but they are considerably (an order of magnitude according to [4]) faster than accessing the Attribute table because of the much larger size of the Attribute table.

A.2 Object API Middle Ware

Exposing the relational database tables to users is both inconvenient and dangerous. First, it takes somebody who is familiar with relational databases as well as our specific database schema to extract the information in a useful way. Second, a small mistake in updating the database directly may leave the database in an incoherent state. The Process Handbook Process Model consists of a set of inter-dependent entities having very complex inter-relationships. A modification to any one component may cause a propagation effect that requires coordinated modifications of dozens of other components in the database.

The solution we have adopted is a three-tier architecture that implements an Object API middleware. The Object API layer abstracts away the relational database and provides to the client an object view of processes. Figure A-2 illustrates the three-tier architecture. The Object API interacts with the relational database through SQL (Structured Query Language) and export objects to clients such as the Process Browser and Editor shown in the diagram.

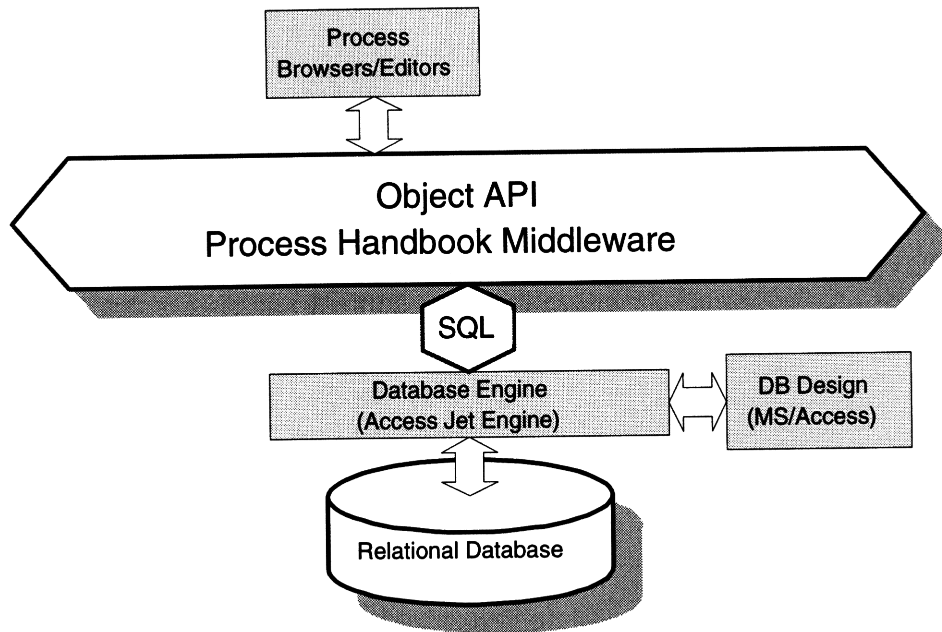


Figure A-2: The three-tier process handbook architecture. (Adapted from [9])

A.2.1 The Purpose of the Object API

Modularity

The Object API makes the Process Handbook implementation more modular. For example, when the size of our process repository grows significantly, we can change the database (currently in Microsoft Access) to an full fledge remote Oracle database and the change will be transparent to any client that is above the Object API level.

Integrity

Another benefit from having the Object API middleware is maintenance of database coherence. The task of modifying an entity in the process database is painstaking. One modification may result in a propagation effect resulting in a dozen other modifications. For example, modifying an activity A will result in new ports for A, creation of a specialization of its parent B (because we have altered a sub-activity in the decomposition of B.), new ports for B, and new connectors. With the Object API middleware, clients can no longer modify the database directly; they must go through

a set of update routines that have been designed to keep database integrity and have been tested for correctness.

Security

Lastly, the Object API middleware makes it possible to implement security. Security becomes an important issue when the size of the user community and the size of the database repository become significantly large. Some form of access control list must be implemented on the process level to restrict access to only the appropriate group of clients. For example, if process P is proprietary to firm A then only clients belonging to the group Firm A should have access to process P. Or, some groups (e.g., administrators, managers) may have the right to modify the processes while other groups (e.g., users, students) have only rights to view the process but not to modify. Since clients have to go through the Object API, security clearance can be performed there on every request. For a more formal discussion of Process Handbook discretionary access control policy, please see [13].

A.2.2 The Objects of the API

The Object API gives an object-oriented view of a relational process database. The object API exports seven first-class objects: PH_DB, Entity, Relation, ObjAttribute, Entities, Relations, and ObjAttributes. Figure A-3 summarizes the properties and methods for API objects.

Notice that there are only five objects in the diagram. That is because the object Collections actually represents three objects: Entities, Relations and ObjAttributes, which are just plurals of the more fundamental objects Entity, Relation and ObjAttribute.

PH_DB

PH_DB is actually a system object rather than an intrinsic object in our process model. PH_DB object allows the client to connect to the database and retrieve data

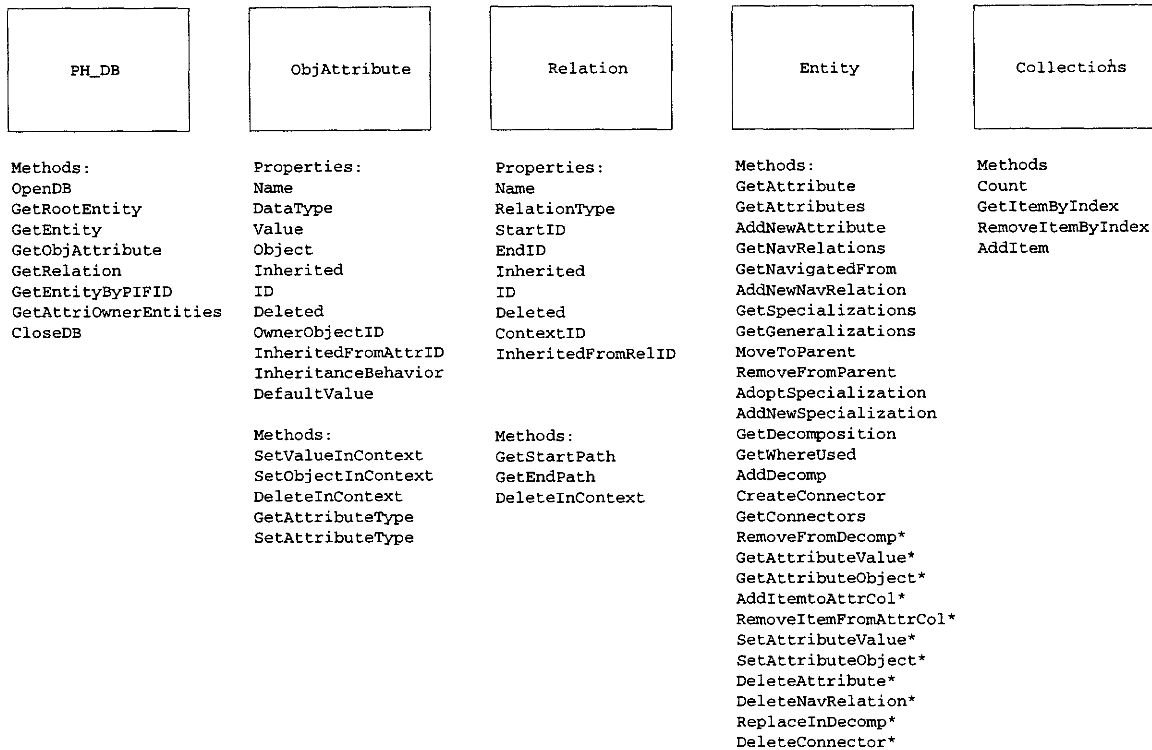


Figure A-3: Properties and methods for API objects (Adapted from [4])

from it.

Entity

Entity object is the most widely used class of the object server. Entity can represent a Thing, Activity, Bundle, Port, Navigational Node, Dependency, Resource, and Attribute-type. All of these can be specialized and decomposed, which makes them an entity. Relations and attributes differ from entity in that they can neither be specialized nor decomposed. In addition, entities exist in their own rights, whereas relations and attributes exist only in the context of an entity.

Relation

A Relation object represents a relationship between two Process Handbook entities. Decomposition relation, connector, and navigational relation can all be represented by relation object. The Inherited property specifies whether the relation is inherited or not. A relation object specifies the two entities that it relates. Therefore, a relation is existence-dependent on two entities. Decomposition relations and navigational relations are directional. But connectors are non-directional because the directional information is already defined by the port type (consumer or producer).

ObjAttribute

An ObjAttribute object represents a Process Handbook attribute belonging to an entity. The DataType property specifies the type (primitive or object) of data stored in this attribute. Like relation, ObjAttribute exists only in the context of an entity.

Collections

The API exports three collections: collection of Entities, collection of Relations, and collection of ObjAttributes. The collection versions of Entity, Relation, and ObjAttribute facilitate the manipulation of sets of objects. Collections can count how many objects they contain (Count), can return objects by index (GetItemByIndex), add new members to the collection (AddItem) and remove a member from the collection (RemoveItemByIndex).

A.2.3 Extensions to the Object API

The above specifications of the Object server is a minimal set of functionality that the Process Handbook guarantees to support. Recent progress has been made to improve the efficiency and power of the Object server.

One of the efficiency improvements is the implementation of caching. The number of children in an entity's specialization and decomposition, or the number of bundles, are now cached. Because operations to compute such information are found to be

expensive and frequent, caching greatly improves access time. Another place that caching is applied is in the entity-bundle-specialization relationship. Extra direct links (relationships) were added between the entity and its specializations so we avoided the extra hop if we are not interested in bundles.

Another extension is in the notion of slots. Each decomposition occupies a slot of its parent entity. Slots assist the implementation of inheritance. Each specializations of an entity automatically inherit the parent entity's slots. For more detailed discussion of the above mentioned extensions and other extensions please see [2].

A.3 Beyond Object API

Once we have a standard interface (Object API) for the Process Handbook, various more user friendly tools can be developed to assist the user in interacting with the complex process database. Figure A-4 illustrate how different components can be layered on top of the Object API middleware.

These tools include graphical process browsers and editors (PH_WIN), web based graphical process viewer server (PH_WEB), report writers, and Process Interchange Format (PIF) Toolkit. Because of the open Object API, remote client or third party clients can be implemented to interact with the Process Handbook objects. Third party clients can even have their own process database that adds to our process repository. Different process interchange format programs can be implemented to convert the process handbook model into other models that can be used to run simulations of the process. However, the two main attractions of the Process Handbook project are still PH_WIN and PH_WEB.

PH_WIN and PH_WEB are the two main interfaces that users can use to interact with the Process Handbook database. The purpose of PH_WIN and PH_WEB interfaces is to provide the user an intuitive and concise graphical representation of the large amount of complex information in the process database. These interfaces are crucial to the theme of the Process Handbook, which is to assist the user in understanding (absorbing large amount of information and visualizing structures

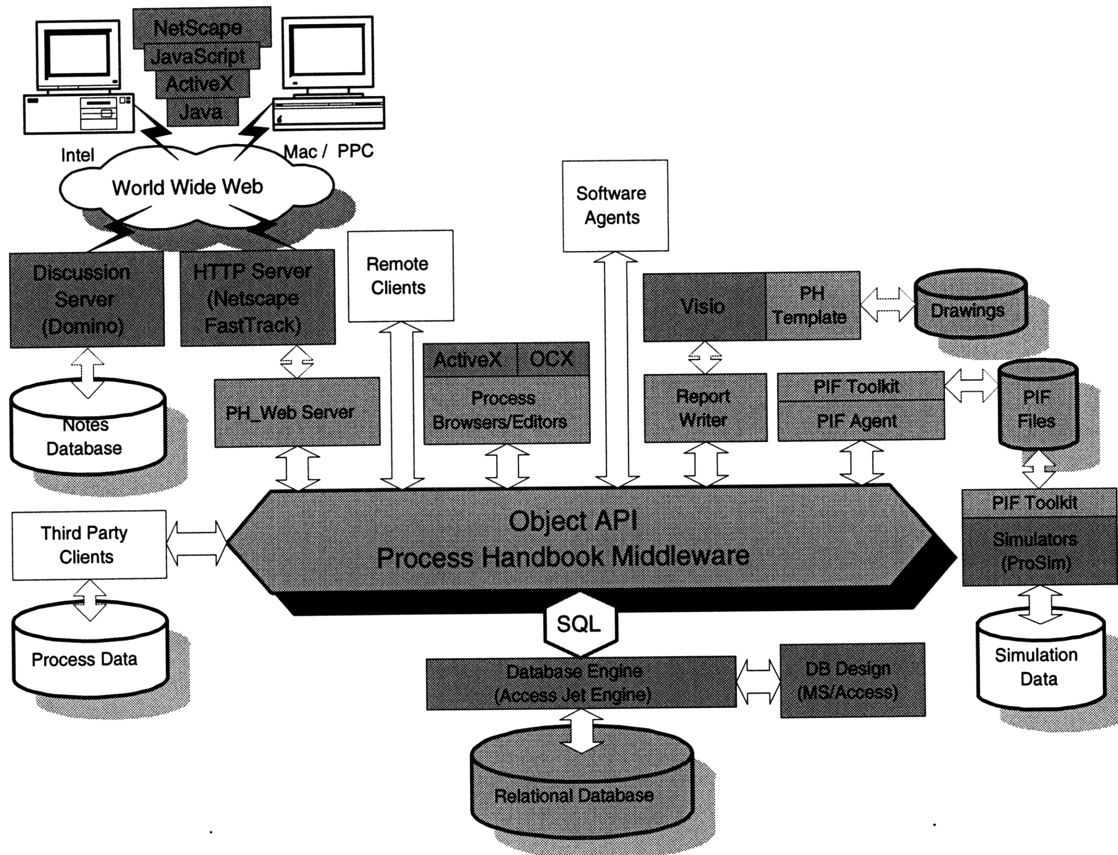


Figure A-4: Components that are built on top of the Object API middleware. (Adapted from [9])

and patterns in seemingly complex and unstructured data) complex processes and to design new processes by modeling after existing processes quickly and efficiently. Designing these interfaces is a challenging task by itself. See [8] for the design PH_WEB and [2] for the design of dependency editor, part of PH_WIN.

A.3.1 PH_WEB

PH_WEB is currently a read-only interface due to the limitations of HTML, CGI, and JavaScript but it has the advantage that it can reach a much larger and more distant audience. With the current web technologies such as Java and ActiveX, editorial capability can be added to PH_WEB in the future. Figure A-5 shows a snapshot of PH_WEB in action. Notice how indentation, positioning, and tables are used to

convey the structure of process hierarchy or decomposition (under the coordination compass).

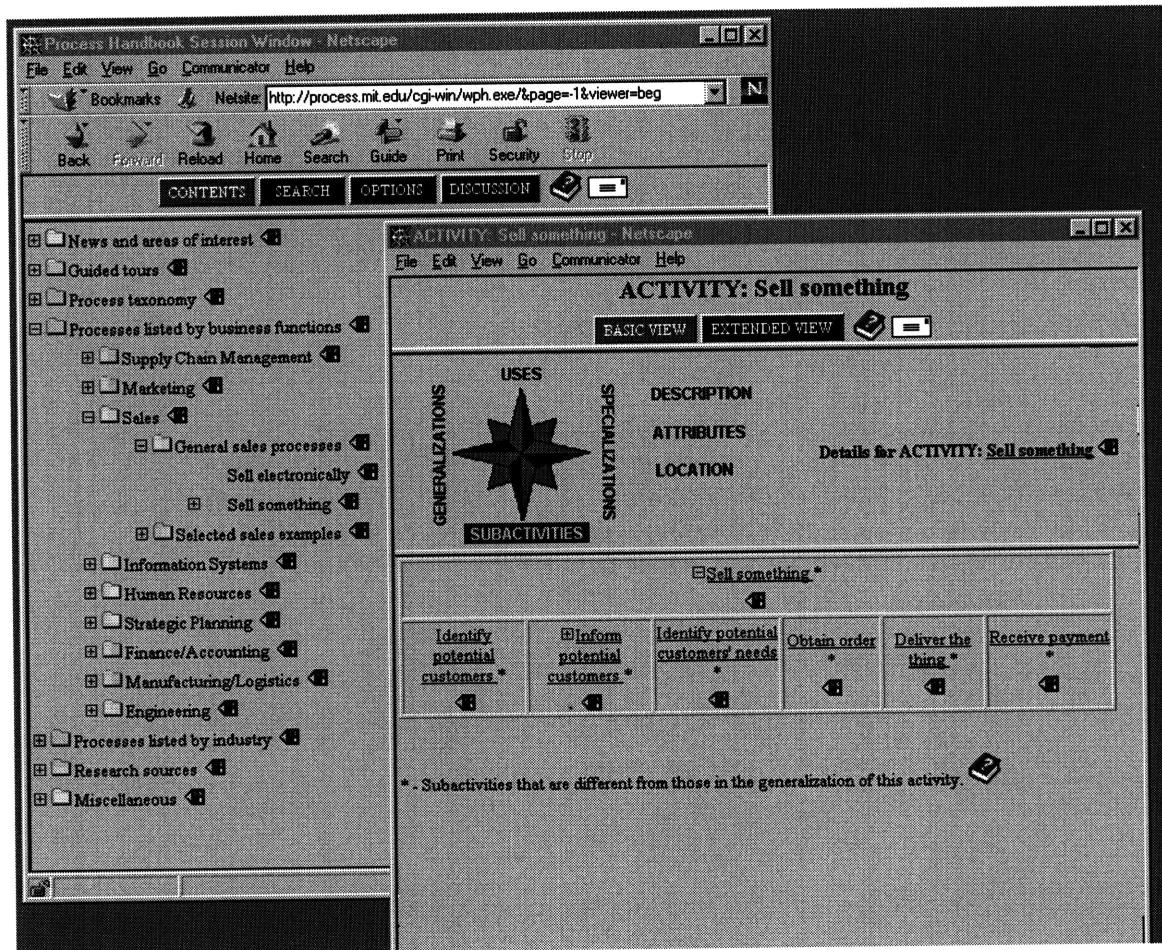


Figure A-5: A snapshot of the Web Process Handbook interface.

A.3.2 PH_WIN

PH_WIN is an application that allows the user to graphically view and modify the process database. PH_WIN has the advantage that it is much more interactive with its drag and drop functionality as well as better graphical representations. PH_WIN is actually a collection of different Process Handbook viewers (decomposition viewer, specialization viewer, activity details viewer, and dependency editor) each lets the user look at the processes in a different angle. Since it is a Windows application, it

has the luxury of displaying lines, arrows, and boxes that can be dragged and modified interactively.

Figure A-6 shows three particular viewers. Notice that the specialization viewer can also display decomposition structures (sell over Internet) at the same time, which is a very convenient and powerful visual metaphor. Also notice the worlds-within-worlds model used in the dependency editor in figure A-6. As the user navigates through different levels of a process's decomposition hierarchy, the context information (make coffee) is preserved. When the user wishes to view the decomposition of a process, the sub-activities of that process are rendered in the containing frame of the parent. This way, both the current level of activities and its context (all of its parents and grandparents) are visible at the same time giving the user a better understanding. Such rendering also allows for easy presentation of vertical connectors. [2]

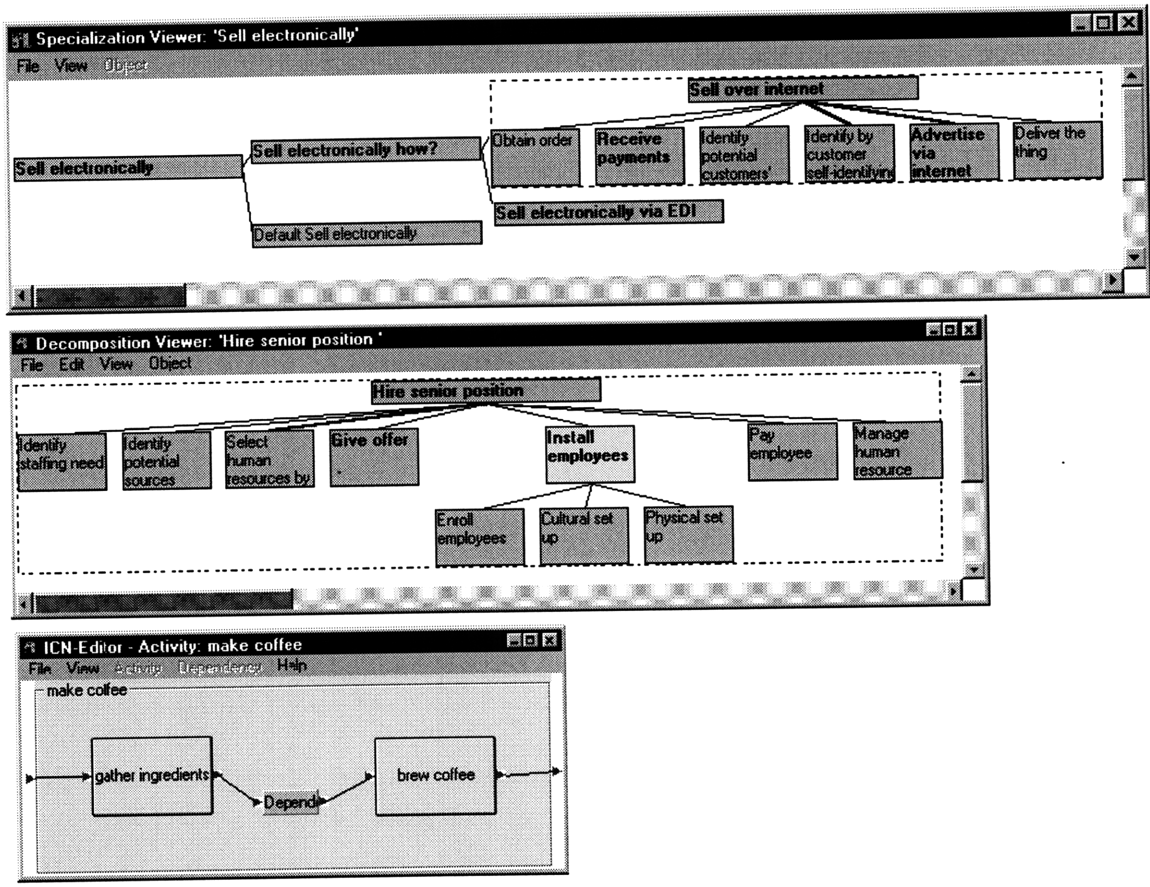


Figure A-6: PH_WIN Viewers and Editors.

Appendix B

Search Algorithms

B.1 Breadth-first Search

Breadth-first search pushes uniformly into the search tree. Breath-first runs a particular query on the entire tree level before going to the next query and expanding the tree to a new level.

Breadth-first Algorithm :

- ▷ Form a one-element binding list consisting of a zero-size binding set.
- ▷ Until we have enough binding sets that satisfy the queries or the binding list is empty,
  - ▷ Remove the first binding set from the binding list; create new binding sets by extending the current binding set to a list of binding sets that satisfy the next query.
  - ▷ Reject the binding set if no new binding sets can be generated from the current one that satisfies the next query.
  - ▷ Add the new binding sets, if any, to the *back* of the binding list.
- ▷ If a binding set satisfying all queries in the query list is found, announce success and continue to find other goal binding sets.

B.2 Depth-first Search

Depth-first search dashes headlong to the bottom of the tree along the leftmost branches. The next step is to back up to the nearest ancestor node that has an unexplored alternative. The algorithm of depth-first search is the same as that for breadth-first search except that newly generated binding sets are placed in the front of the list instead of the back.

Depth-first Algorithm :

- ▷ Form a one-element binding list consisting of a zero-size binding set.
- ▷ Until we have enough binding sets that satisfy the queries or the binding list is empty,
  - ▷ Remove the first binding set from the binding list; create new binding sets by extending the current binding set to a list of binding sets that satisfy the next query.
  - ▷ Reject the binding set if no new binding sets can be generated from the current one that satisfies the next query.
  - ▷ Add the new binding sets, if any, to the *front* of the binding list.
- ▷ If a binding set satisfying all queries in the query list is found, announce success and continue to find other goal binding sets.

When you know that all partial paths either reach dead ends or become complete paths after a reasonable number of steps, then depth-first is a good strategy. On the other hand, depth-first search is a bad idea if there are long paths, even infinitely long paths, that neither reach dead ends nor become complete paths. This situation does not occur in PQL queries. In addition, the depth of any branch of a PQL search tree is at most the number of statements in the PQL query. The fact that PQL search trees are usually shallow (size of the PQL query) and fat (branching factor) makes depth-first search very convenient. Breath-first search is that it can handle trees with infinitely deep or effectively infinitely deep trees; an advantage that we don't need.

Note that breath-first search is a bad idea if the branching factor is large or infinite, because of exponential explosion, which is very likely in PQL queries.

Depth-first search has one additional advantage; it can provide partial results as the search proceeds. This property allows for the option of terminating the search after a specified number of matches are found. Because depth-first considers a possible path to completion before considering others, it can save on memory by freeing the visited paths before expanding new ones to visit. A breadth-first search, on the other hand, will keep expanding until the final level. Because of the high branching factor nature of PQL queries, depth-first search is a better strategy than breadth-first search. Breadth-first search can be used to show how intermediate results are passed from query to the next query, which can provide useful information.

Appendix C

PQL Implementation Notes

C.1 Overall Design

PQL is implemented in Visual Basic 5, a language that is used to implement other Process Handbook application as well. The implementation of PQL is divided up into objects and modules. The objects are PDB (Process Handbook database object), Pair (a binding object), BSet (a binding set object), BList (a binding list object), QList (a query list object) and generic (a generic entity object). The modules are Search (search and database query functions), Parser (interpreter routines for PQL syntax), List (list processing functions), Operators (logical operator functions) and Interface (a simple user interface for PQL).

It is useful to understand the flow of PQL logic. Figure C-1 shows how PQL execution would flow through the different modules. Execution starts with the user entering a query through the interface. The interface is not considered as a parts of the core PQL. Other interfaces can be built to interface with PQL.

First, the query is pass through a preliminary parser that parses the PQL query into different structural parts (SELECT clause, FROM clause, and sub-queries if logical operators are used). Then the query is passed into the PQL tree search engine. In the process of expanding the search tree, PQL looks at each primitive queries in turn and dispatches to the correct syntax parser base on the type of the primitive query. After parsing the syntax, the primitive query is processed by their respective

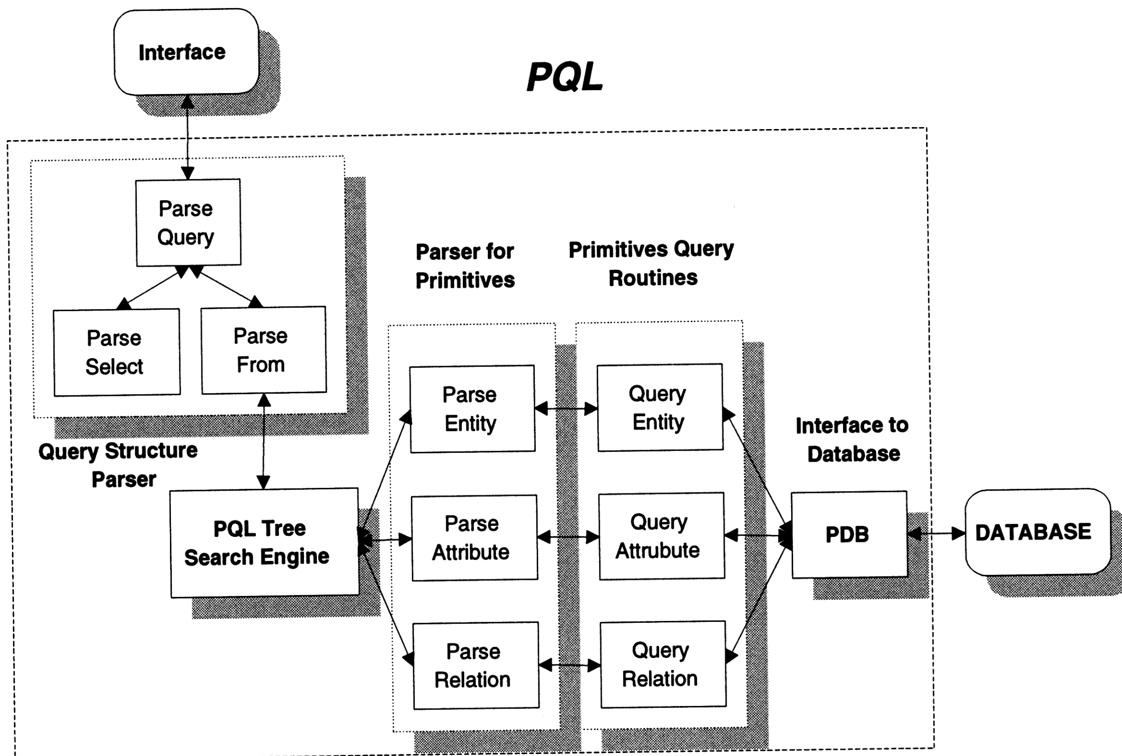


Figure C-1: PQL Implementation Modules

query routines. These routines access the Process Handbook database through the PDB interface. Results of primitive queries are passed back to the tree search engine to build new branches. Before the matches for the query are passed back to the interface, it is logically operated if there are any logical operators (AND, OR) in the query and then projected according to the SELECT clause. The rest of the appendix give the specifications of the major PQL objects and modules.

C.2 PQL Objects

C.2.1 PDB

The *PDB* object is the interface to the underlying Process Handbook database.[1] CreateSnapshot( ) is the only method that can access the contents in the database, taking an SQL string as parameter and returning a Recordset of results. The method is read-only, thus PQL does not modify the database.

Data

Private db As Database

Methods

```
Public Function OpenDB(ByVal dbpath As String) As Boolean
    Modifies: Self
    Function: Opens database specified by dbpath.
              Returns TRUE if successful and FALSE otherwise.

Public Sub CloseDB()
    Modifies: Self
    Function: Closes database db.

Public Function CreateSnapshot(SQLstr As String) As Recordset
    Function: Returns a snapshot containing the results of the
              SQL query string SQLstr.
```

C.2.2 Pair

The *Pair* object represents a binding pair. If b is a binding pair (object value) then we say that b.Obj (object) is bounded to b.Val (value). In PQL, the binding pair (?a 3122) is used to represent the binding of the PQL variable ?a to the entity ID 3122.

[1] Currently a relational database implemented using MS Access

Data

Public obj As String Public Val As Long ' Variant takes up too much space.

Methods

```
Public Function ToString() As String
    Function: Returns the itself in string form of "(obj val)".
```

C.2.3 BSet

BSet object represent a set of bindings (or a binding set). An example of a binding set is ((?a 534) (?b 3122) (?c 3349)). In PQL, a binding set represents one particular instance of the world or environment. In this instance of the world, ?a is bounded to the process with ID 534, ?b is 3122, and ?c is 3349. In PQL a match or a partial match is represented by an binding set. We also make queries that are conditioned on a pre-existing binding set or world.

We would like *BSet* to inherit from *Collection* so as to make use of collection's list like function (e.g., iterator) However, Visual Basic does not support inheritance. We get around this deficiency (at the risk of breaking encapsulation) by making *BSet* data, *Bindings*, public, hence exporting the useful *Collection* methods.

Data

Public Bindings As Collection

Methods

```
Public Function Count()
    Function: Returns the number of elements in the binding set.

Public Function Add(ByVal obj As String,
                    ByVal Val As Variant) As Boolean
```

```
    Modifies: the binding set
    Function: Adds the binding (obj val) to the itself.
              Returns TRUE if obj is not already bound in the
              set and FALSE otherwise.

Public Function Remove(obj As String) As Boolean
    Modifies: Self
    Function: Remove the binding for obj from itself.
              Returns TRUE if obj is bound in the set before
              removing and FALSE otherwise.

Public Function IsEmpty() As Boolean
    Function: Returns TRUE if the self has no binding and
              FALSE otherwise.

Public Function IsBinded(var As String) As Boolean
    Function: Returns TRUE if the object var is bound in the
              binding set and FALSE otherwise.

Public Function Equals(b_set As Bset) As Boolean
    Function: Compares another binding set b_set with itself.
              Returns TRUE if they have the same bindings and
              FALSE otherwise.

Public Function ToString() As String
    Function: Returns a string version of itself.

Public Function ToNameString() As String
    Function: Same as ToString() but instead of returning a list of
              (obj ent_id), it returns a list of (obj ent_name ent_id),
              where the ent_id is used to looked up the corresponding
              ent_name in the database.

Public Function getValue(ByVal obj As String) As Long
    Requires: obj is bound in the binding set.
    Function: Returns the value that is bound to the object obj.

Public Function GetPair(obj As String,
                        bpair As Pair) As Boolean
    Modifies: bpair
    Function: bpair is set to the binding pair of obj.
              Returns TRUE if obj is bound in the binding set and
              FALSE otherwise.

Public Function Copy() As Bset
```

```
    Function: Returns identical but distinct BSet of itself.

Public Function Append(b As BSet) As Boolean
    Modifies: the binding set
    Function: Merge the bindings of BSet b to itself.
              Returns TRUE if there is no conflicts in bindings and
              FALSE otherwise.

Public Function Expand(b As BList) As BList
    Function: Returns a BList that is the Cartesian product of a
              BSet (Self) and BList b.  The resulting elements of
              BList is formed by taking each binding set of b and
              merging with self.
```

C.2.4 BList

BList object represent a list of binding sets (or binding list). An example of a binding list containing two binding sets is (((?a 534) (?b 3122) (?c 3349)) ((?a 534) (?b 3140) (?c 6074))) In PQL, a binding list represents a collection of matches or partial matches: a collection of all the possible world in which the PQL query is satisfied. Just like *BSet*, we want *BList* to inherit from *Collection*. Therefore, the *List* in BList is public.

Data

Public List As Collection

Method

```
Public Function Append(b_list As BList) As BList
    Function: Returns a BList that is the result of appending
              another BList b_list to itself.

Public Function ToString() As String
    Function: Returns a string description of the itself.

Public Function ToNameString() As String
```

```
    Function: Same as ToString() but the ent_name is looked up in
              the database from the ent_id and inserted into the string.

Public Sub Create(obj As String, values As Collection)
    Modifies: Self
    Function: Create SELF (BList) from a obj and a list of values.
              SELF = (((obj val1)) ((obj val2)) ... ((obj valN)))

Public Sub Create_from_IDArr(obj As String, ByRef IDArr() As Long)
    Modifies: Self
    Function: Same as Create() but the list of values is in the form
              of an array instead of a collection.

Public Function IsEmpty() As Boolean
    Function: Returns TRUE if self is empty and FALSE otherwise.

Public Function RemoveDuplicates() As BList
    Function: Returns a BList that contains distinct BSets.
```

C.2.5 QList

QList object represent a list of strings (or query list because the strings are primitive queries). An example of a query list is ((LET ?a BE identify potential customers) (LET ?b BE Sell over internet*) (REL ?a has-spec* ?c) (REL ?c is-used-by* ?b)) In PQL, a query list is used to store the query to be process by the tree search. We want *QList* to inherit from *Collection*; so *List* in QList is public.

Data

Public List As Collection

Method

```
Public Function First() As String
    Function: Returns the first element of the self.
```

```
Public Function Rest() As QList
    Function: Returns ANOTHER QList that is SELF without the
              first element.

Public Function ToString() As String
    Function: Returns a string description of self.

Public Function IsEmpty() As Boolean
    Function: Returns TRUE if self is empty and FALSE otherwise.

Public Function Append(q_list As QList) As QList
    Function: Returns a QList that is q_list appended to self.
              Usage: c = a.Append(b)

Public Sub Parse(s As String)
    Modifies: Self
    Function: Parses a string of queries into Self (QList)
              where each query is delimited by a semicolon (;)
              and // are used for comments.
```

C.3 PQL Functions

C.3.1 Parser

The *Parser* module is a collection of functions that interprets the PQL syntax. Top level function Dispatch() looks at a primitive query and dispatches to the correct handler for the query according to the primitive query type (entity, attribute, or relation type). Functions, Parse_Ent_Name(), Parse_Attrib(), and Parse_Relation are each responsible for processing their respective types of primitive PQL query. Parse_Select parses the SELECT ... FROM clauses of the PQL query. Other functions are significant helper functions for the parser.

```
Public Function Dispatch(ByVal Query As String,
                         b_set As Bset,
                         new_b_list As BList) As PQL_Return_Type
    Function: Dispatches the query to the appropriate handler.
```

```
            Relays the return of the resulting processing procedure.

Public Function PQL_Classify(token_list As Collection) As PQL_Type
    Function: Returns the PQL Query Type given a list of query tokens
              PQL Queries can be one of the following type and have
              the forms:
                ENT_TYPE:          (ENT ?x is <name> )
                ATTRIB_TYPE:       (ATTRIB ?x ... )
                RELATION_TYPE:     (RELATION ?x <relation> ?y)
                INVALID_TYPE:       Not one of the above.

Public Function Parse_Ent_Name(token_list As Collection,
                               b_set As Bset,
                               new_b_list As BList) As PQL_Return_Type
    Function: Performs the primitive ENT query in the environment of
              BSet b_set and returns the result as new_b_list.
              Returns PQL_MATCH if query is satisfied,
                      PQL_NO_MATCH if query is not satisfied,
                      PQL_SYNTAX_ERROR if query is not in the form of
                           (ENT ?x is <entity_name>)
                      PQL_UNKNOWN_ERROR if there is an internal error.

Public Function Parse_Attrib(token_list As Collection,
                             b_set As Bset,
                             new_b_list As BList) As PQL_Return_Type
    Function: Performs the primitive ATTRIBUTE query in the environment
              of BSet b_set and returns the result as new_b_list.
              Returns the same protocol as Parse_Let_Name() and syntax
              error if query is not in the form of:
                  (ATTRIB ?x <attrib_name> <op> <value>)

Public Function Parse_Relation(token_list As Collection,
                               b_set As Bset,
                               new_b_list As BList) As PQL_Return_Type
    Function: Performs the primitive RELATION query in the environment
              of BSet b_set and returns the result as new_b_list.
              Returns the same protocol as Parse_Let_Name() and syntax
              error if query is not in the form of:
                  (RELATION ?x <relation_type> ?y)
                       where <relation_type> is one of
                          has-decomp    has-decomposition
                          is-used-by    has-comp
                          has-spec      has-specialization
                          has-gen       has-generalization
```

```
Public Function Tokenize(ByVal Query As String,
                            separator As String) As Collection
    Function: Returns an ordered collection of tokens from the query
              string using the token separator "separator".

Private Function IsVar(var As String) As Boolean
    Function: Returns TRUE if var is in the form of a PQL variable
              and FALSE otherwise.  PQL variables begins with a "?".

Public Function Parse_Select(s As String,
                            sel_vars As Collection) As PQL_Return_Type
    Function: Parses the SELECT clause and returns the variables to be
              selected in a collection sel_vars.
              Returns PQL_Return_Type.

Public Function Parse_Query(Query As String,
                            b_list As BList) As PQL_Return_Type
    Function: High level parse query.

Public Function Find_Matching_Parenthesis(s As String) As Integer
    Requires: String s begins with a left parenthesis "(".
    Function: Returns the position of the matching right parenthesis.

Public Sub Filter_CRLF(s As String)
    Function: Get rid of all CRLF characters and replace them
              with space
```

C.3.2 List

The *List* module is a collection of list processing routines that are useful in PQL's search engine. They deal with the converting of database query results into lists and arrays that can be processed by the PQL search engine.

```
Public Function Recordset_to_BList(vars As Collection,
                                   snap As Recordset) As BList
    Function: Returns a BList (a list of binding sets) from a
              snapshot recordset and a list of variables to bind to.
              If vars = (V1, V2, V3, ..., Vn) and snap Recordset is a
              two dimensional table with columns denoted by
```

```
            (C1, C2, C3, ..., Cm) where n <= m then the returned
            BList is a list containing element i is a
            BSet = ((V1 C1) (V2 C2) (V3 C3) ... (Vn Cn)) for row i
            in our snap Recordset.

Public Function Snapshot_to_Array(
                        ByRef MySnap As Recordset,
                        ByRef IDs() As Long) As PQL_Return_Type
    Function: Same as Recordset_to_BList but instead of output it
            to an BList, it outputs it into an Array.

Public Sub RemoveDuplicates(ByRef IDArr() As Long)
    Modifies: IDArr()
    Function: Takes in an array of ID's and removes the duplicates
            in IDArr.
```

C.3.3 Search

The *Search* module contains the search engine for PQL. Breadth_First() and Depth_First() are two ways to perform a PQL tree search. Currently, PQL is only using Depth_First() because it is more efficient.[2] The others routines deal with specific types of primitive queries as their name indicates.

```
Public Function Breadth_First(q_list As QList,
                        return_list As BList) As PQL_Return_Type
    Function: Takes a query list q_list and returns a binding list
            return_list such that every binding set in return_list
            satisfies the query expressed in q_list.  It uses a
            iterative breath_first search algorithm.

Public Function Depth_First(q_list As QList,
                        return_list As BList) As PQL_Return_Type
    Function: Takes a query list q_list and returns a binding list
            return_list such that every binding set in return_list
            satisfies the query expressed in q_list.  It uses a
            iterative depth_first search algorithm.
```

[2]See appendix B for discussion of search algorithms.

```
Public Function queryName(var As String,
                          name As String,
                          b_set As Bset,
                          new_b_list As BList) As PQL_Return_Type
    Function: Given a pre-existing BSet b_set and variables var
              and string name, finds and makes a BList new_b_list of
              BSets of var where var has entity name matching the
              string name.

Public Function queryAttrib(var As String,
                            attrib_name As String,
                            attrib_op As String,
                            attrib_value As String,
                            b_set As Bset,
                            new_b_list As BList) As PQL_Return_Type
    Function: Given a pre-existing BSet b_set and variables var,
              string attrib_name, attrib_op, and attrib_value,
              finds and makes a BList new_b_list of BSets of var
              where var has attribute satisfying the predicate
              attrib_name <attrib_op> attrib_value.

Public Function querySpec(var1 As String,
                          var2 As String,
                          b_set As Bset,
                          new_b_list As BList,
                          recursive As Integer) As PQL_Return_Type
    Function: Given a pre-existing BSet b_set and variables var1
              and var2, finds and makes a BList new_b_list of BSets
              of var1 and var2 where var1 has var2 as specialization
              within recursive number of levels in the Process
              Handbook hierarchy.

Public Function queryDecomp(var1 As String,
                            var2 As String,
                            b_set As Bset,
                            new_b_list As BList,
                            recursive As Integer) As PQL_Return_Type
    Function: Given the environment of BSet b_set and variables var1
              and var2, finds and makes a BList new_b_list of BSets
              of var1 and var2 where var1 has var2 as decomposition.
```

Bibliography

[1] H. Abelson and G. Sussman. *Structure and interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[2] Zia Ahmed. An integrated dependency editor for the process handbook. Technical report, MIT Department of Electrical Engineering and Computer Science, 1998.

[3] Francois Bancilhon, Claude Delobel, and Paris Kanellakis. *Building an Object-Oriented Database System, The Story of O_2.* Morgan Kaufmann Publishers, 1996.

[4] Umar Farooq. An object server for the process handbook. Technical report, MIT Department of Electrical Engineering and Computer Science, 1997.

[5] Henry Korth and Abraham Silberschatz. *Database System Concepts.* McGraw-Hill, Inc, 1991.

[6] Thomas Malone, Kevin Crowston, Jintae Lee, Brian Pentland C̃hrysanthos Dellarocas, George Wyner, John Quimby, Charley Osborne and Abraham Bernstein. Tools for inventing organizations: toward a handbook of organizational processes. Technical report, MIT Center for Coordination Science, 1997.

[7] Thomas Malone, Kum-Yew Lai, and Christopher Fry. Experiments with oval: a radically tailorable tool for cooperative work. Technical report, MIT Center for Coordination Science, 1995.

[8] Vuong Nguyen. A web based interface metaphor for representing complex process knowledge. Technical report, MIT Department of Electrical Engineering and Computer Science, 1997.

[9] John Quimby. Process handbook program review. Technical report, MIT Center for Coordination Science, 1997.

[10] Marc Scholl and H. J. Schek. Survey of the cocoon project. Technical report, ETH Zurich Department of Computer Science, Information Systems, 1992.

[11] Markus Tresch and Marc Scholl. Implementing an object model on top of commercial database systems. Technical report, ETH Zurich Department of Computer Science, Information Systems, 1992.

[12] Patrick Henry Winston. *Artificial Intelligence. Third Edition.* Addison-Wesley Publishing Company, 1992.

[13] Calvin Yuen. A discretionary access control policy for the process handbook. Technical report, MIT Department of Electrical Engineering and Computer Science, 1997.