A Data Abstraction with Inheritance in the Process Handbook

by

Erfanuddin Ahmed

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of

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

at the Massachusetts Institute of Technology
May 1995

© 1995 M.I.T. All rights reserved.

Author

Department of Electrical Engineering and Computer Science
May 30, 1995

Certified by

Professor Thomas W. Malone
Thesis Supervisor

Accepted by

Chairman, Department Committee on Graduate Theses

AUG 10 1995
A Data Abstraction with Inheritance
in the Process Handbook

by

Erfanuddin Ahmed

Submitted to the
Department of Electrical Engineering and Computer Science

May 30, 1995

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Process Handbook is a project at the MIT Center for Coordination Science that involves the development of an on-line repository of business process descriptions at various levels of specialization. In the implementation of the Process Handbook, a three tier client server architecture was employed. This thesis describes the design and implementation of the middle tier as a data abstraction that allows viewers and editors to access and modify data in terms of the Process Handbook methodology of business process representation. This data abstraction has been implemented as an Application Programmers Interface (API) such that all Process Handbook viewer and editor applications can be developed with this API independent of the specific implementation of the relational database that is used as the storage device. Moreover, the data abstraction is responsible for supporting the Process Handbook notions of inheritance. Viewers and editors are transparent to the fact that the inheritance mechanism is being provided on top of a relational database. The implementation of the data abstraction was based on the specific requirements of the Process Handbook and has provided for a data abstraction that is unique from other existing middle-tier data abstractions.

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

I wish to thank a number of people who have assisted me in various ways in the writing of my thesis. I would like to start by thanking Professor Thomas Malone, my thesis supervisor, for his encouragement and continual support of my research. I am grateful to my friends and fellow research assistants, namely, Frank Chan, Naved Khan and Narasimha Rao who were (un)fortunate enough to use the middleware data abstraction through its incremental evolution of different versions. I truly appreciate their feedback and discovery of bugs. My thanks to the all those at the MIT Center for Coordination Science, who have helped me in various ways throughout my year-long tenure at the Center. Last but not least, I would like to convey my heartfelt thanks and sincere appreciation to my parents for standing by me through all these years. I could not have made it without their commitment and concern for my education. Finally, my special thanks to my siblings who have always guided me in my endeavors.
Contents

1. INTRODUCTION ............................................................................................................. 13

2. BACKGROUND ............................................................................................................. 15

2.1 LITERATURE ............................................................................................................. 15

2.2 PROCESS HANDBOOK ............................................................................................. 17

2.2.1 Inheritance ......................................................................................................... 17

2.2.2 Bundles ............................................................................................................... 21

2.2.3 Coordination ....................................................................................................... 21

2.2.4 Context ................................................................................................................. 22

3. SYSTEM OVERVIEW ................................................................................................. 24

3.1 SYSTEM REQUIREMENTS ....................................................................................... 24

3.1.1 Process Representation and Storage ................................................................. 24

3.1.2 Inheritance ......................................................................................................... 24

3.1.3 Multiple Viewers/Editors .................................................................................. 25

3.1.4 Translation .......................................................................................................... 25

3.1.5 Scaleable Design ................................................................................................ 26

3.1.6 Practical Considerations ..................................................................................... 26

3.2 SYSTEM DESIGN: THREE TIER ARCHITECTURE ................................................... 27

3.2.1 Description .......................................................................................................... 27

3.2.2 Advantages ......................................................................................................... 28

3.2.3 Disadvantages ..................................................................................................... 29

3.3 OTHER PROTOTYPE IMPLEMENTATIONS ............................................................... 30

3.3.1 Kappa PC Implementation ................................................................................. 30

3.3.2 Lotus Notes Implementation .............................................................................. 30

3.3.3 Envision Implementation .................................................................................... 31

4. DATABASE DESIGN ................................................................................................... 32

4.1 RELATIONAL DATABASE VS. OBJECT-ORIENTED DATABASE ......................... 32
4.2 DATABASE DESIGN ..............................................................................................................33

5. MIDDLEWARE ..................................................................................................................36

5.1 MIDDLEWARE API ........................................................................................................37

5.1.1 Description ...............................................................................................................37

5.1.2 Design & Implementation Considerations ...................................................................37

5.1.3 User Guide .................................................................................................................40

5.2 INHERITANCE ..............................................................................................................42

5.2.1 Attribute Inheritance ...............................................................................................43

5.2.1.1 Description ..........................................................................................................43

5.2.1.2 Design and Implementation Considerations ..........................................................44

5.2.1.3 Example ..............................................................................................................47

5.2.2 Decomposition Inheritance .......................................................................................49

5.2.2.1 Description ..........................................................................................................49

5.2.2.2 Design and Implementation Considerations ..........................................................50

5.2.2.3 Example ..............................................................................................................52

5.2.3 Dependency Inheritance ............................................................................................52

5.2.3.1 Description ..........................................................................................................52

5.2.3.2 Design and Implementation Considerations ..........................................................54

5.2.3.3 Example ..............................................................................................................54

6. FUTURE WORK .............................................................................................................57

7. CONCLUSION ...............................................................................................................59

REFERENCES ......................................................................................................................61

APPENDIX A: MIDDLEWARE API SPECIFICATIONS .........................................................63
List of Figures

Figure 1: An example class hierarchy ................................................................. 18
Figure 2: Relationship between specializations and decompositions ...................... 20
Figure 3: An example of bundles ........................................................................ 21
Figure 4: The Process Handbook architecture ...................................................... 27
Figure 5: Entity relationship diagram .................................................................... 35
Figure 6: Relationship between API module and inheritance module ...................... 36
Figure 7: Relationship between slot-pointer and value-pointer in a specialization
  hierarchy ........................................................................................................... 46
Figure 8: Example specializations of “Sell Product” .............................................. 47
Figure 9: Implementation of decomposition inheritance ...................................... 50
Figure 10: Example of context mechanism ............................................................ 52
Figure 11: Example decompositions before addition of dependency ...................... 55
Figure 12: Example decompositions after addition of dependency ........................ 56
List of Tables

Table 1: Examples of common dependencies between activities and alternative coordination processes for managing them..........................................................22
Table 2: Middleware API functions related to activities.................................................40
Table 3: Middleware API functions related to bundles...................................................41
Table 4: Middleware API functions related to dependencies........................................42
Table 5: Value of attribute "Cost" at subsequent steps..................................................48
1. Introduction

The Process Handbook is a project at the MIT Center for Coordination Science that involves the development of an on-line repository of business process descriptions at various levels of specialization (Malone, et al., 1993). The handbook has a methodology for business process representation which is rich enough that once a process is represented with this methodology, it can be viewed in a variety of other common process formats including, but not limited, to IDEF0, data flow, etc. The methodology used explicitly represents the similarities and the differences among a collection of related business processes. With this approach, it is possible to first represent a generic “process” and then represent specializations of this process by indicating how they differ from the generic case. At each level of specialization, processes can inherit attributes from their parents and add or change attributes of their own. In addition to specializations, each generic process (i.e., activity) can be decomposed into sub-processes (i.e., subactivities), and specializations of this generic process will inherit its decompositions. Finally, the methodology for representing business processes uses the notion from coordination theory that coordination processes can be thought of as ways of managing dependencies among activities.

The overall architecture for the Process Handbook is a three-tier client server architecture in which the first tier is a relational database, the middle tier is a “Process Handbook Logic” layer, and the third tier is applications that view/edit data from the repository. This thesis designs and implements the “Process Handbook Logic” layer as a data abstraction that allows for modular data access and modification of processes in the repository in terms of the Process Handbook methodology of business process representation. This data abstraction has been implemented as an Application Programmers Interface (API) to the underlying database such that all Process Handbook viewer/editor applications can be developed with this API independent of the specific implementation of the database repository. In developing the API, the specific requirements of the Process Handbook were taken into consideration to provide for a data
abstraction that is unique from other existing middle-tier data abstractions. Also, since a relational database was used in the overall Process Handbook architecture, it was necessary to have an inheritance engine within the middleware data abstraction to support the Process Handbook notions of inheritance within the specialization hierarchy.

Background information on previous work related to the thesis topic and on the Process Handbook is supplied in Section 2. An overview of the Process Handbook system architecture is described in Section 3, and the database design is provided in Section 4. The middle-tier data abstraction including the API and the inheritance mechanisms is discussed in Section 5. Finally, future work and conclusions are provided in Section 6 and Section 7 respectively.
2. Background

2.1 Literature

In this section, previous work related to building an object interface on top of a relational database is discussed. The works discussed below are indicative of the different approaches that have been taken.

The Iris database system (Fishman, 1989; Lyngbaek and Kent, 1986) is a research prototype at Hewlett Packard Laboratories in Palo Alto that consists of an object manager that was built on top of a relational database system. The Iris database system has an extended SQL language that allows object-based manipulation. This extended language called Object-SQL (OSQL) (Beech, 1988) allows users to define new classes into a class hierarchy and provides for object identity and user-defined functions.

The POSTGRES system (Stonebraker and Rowe, 1986) is an extended relational database at the University of California at Berkeley that provides objects, object identifiers, compound objects, multiple inheritance, versions, history data, and a powerful extended relational query language to fetch and manipulate data. The query language, called POSTQUEL includes a transitive closure operation, object identifiers, POSTQUEL-defined attributes, user-defined primitive types and access methods, triggers, inferencing, a portal mechanism to refer to records from multiple relations, and a relation type hierarchy.

Scholl and Schek propose that allowing nested relation schemes to be recursively defined within the relational model yields the necessary flexibility with respect to structure to allow a relational model to be compatible with object-orientation (Scholl and Schek, 1990). These nested relations provide the complex features demanded by object models and provide a powerful query language to exploit the complex data structure while keeping the advantages of the declarative, set-oriented paradigm.
(Keim, et al., 1993) propose a schema transformation and a query translation algorithm to allow for object-oriented querying of existing relational databases. The schema transformation uses additional semantic information to enhance the relational schema and transform it into a corresponding object-oriented schema. If the additional semantic information can be deduced from an underlying entity-relationship design schema, the schema transformations may be done automatically. To query the created object-oriented schema requires the use of Structured Object Query Language (SOQL) which provides declarative query facilities on objects. The query translation algorithm automatically translates SOQL queries into equivalent SQL queries for the original relational schema.

Brodie uses ideas from artificial intelligence, programming languages and databases to provide for a conceptual modeling of complex, object-oriented applications (Brodie 1980). The methodology is based on extending a relational framework using semantic network, predicate calculus and actor concepts from artificial intelligence and more importantly, data, procedure and control abstractions from programming languages. This extended model supports hierarchies of multiply typed data abstractions needed for the intended application.

Lo and Wen propose an object-oriented Management Information Base (MIB) based on existing relational database management system technology (Lo and Wen, 1992). MIB is a repository of network management information developed by the International Standards Organization (ISO) and is generally viewed as object-oriented. Their basic idea is to buffer the relevant object data which is managed by an object-oriented database running in memory, and store them back to the relational database as needed. This allows for quick access to object data while storing them in a relational database to acquire persistence. Applications are transparent to the fact that the repository services are being provided by a relational database.
2.2 Process Handbook

The Process Handbook is a project at the MIT Center for Coordination Science that involves the development of an on-line repository of business process descriptions at various levels of specialization (Malone, et al., 1993). This on-line tool can be used to provide the relative advantages and disadvantages of alternative business processes. The uses of the handbook are several fold. The handbook is intended to help in the redesign of existing business processes, in the invention of new business processes, especially those enabled by information technology, and in the automatic generation of software to support and analyze business processes, e.g. simulation, workflow automation tools.

A key feature of the Process Handbook is its representation of business processes. The Process Handbook has a methodology of representing business processes that explicitly represents the similarities and the differences among a collection of related business processes. This methodology uses ideas of inheritance from computer science and concepts about managing dependencies from coordination theory.

2.2.1 Inheritance

Simple inheritance, as applied in the object-oriented realm of computer science, is a mechanism by which objects organized in an increasingly specialized object hierarchy inherit properties from their parents. Everything that is true of a parent is true of its subclasses, effectively meaning that every instance of a subclass is also a member of its parent classes. For example in Figure 1\(^1\), everything that holds true for “Employee” also holds for “Engineer”, but not vice-versa. An extension to this simple inheritance is the idea of multiple inheritance by which a subclass can have more than one parent. For example in Figure 1, the object “Software Engineer Consultant” is a subclass of both “Software Engineer” and “Consultant” and thus by multiple inheritance, everything that holds true for “Consultant” and “Software Engineer”, also holds true for “Software Engineer Consultant”.

\(^1\) Adapted from Brown, A. (1991), Fig 2.1.
In the Process Handbook, a business processes is described by its decomposition of subactivities and the necessary coordination mechanisms required to manage this process. The decomposition of a business process is similar to the notion of aggregation in computer science (Kim & Lochofsky, 1989) where several objects can be aggregated together to form a new object. For example, in computer science, the object “Car” can be an aggregation of the objects “Body”, “Frame”, and “Wheels”; similarly, in the Process Handbook, the process “Sell Product” can be decomposed into “Identify prospects”, “Inform prospects about product”, “Obtain order”, “Deliver product” and “Receive payment.”

The processes in the Handbook are organized in an increasingly specialized hierarchy. For example, the process “Mail order sales” can be specialized into “Mail order book sales” and “Mail order clothing sales”, and “Mail order clothing sales” can be further specialized into “Lands End” and “L. L. Bean”. The organization of processes within the Handbook also allows for multiple parents in the specialization hierarchy. For example, the activity “Mail order clothing sales” can be a specialization of both “Mail order sales” and “Clothing sales”. Each specialized process inherits the attributes of its parents, the decompositions of its parents and also the coordination mechanisms. At each level of specialization, attributes, decompositions and coordination processes can be modified,
added or deleted as appropriate for the specialized activity. Inheritance in the Process Handbook is somewhat different than the traditional computer science notions of inheritance discussed above. In the traditional computer science notion of inheritance, everything that is true in a parent must be necessarily true in its subclasses, whereas in the Process Handbook this fact does not always necessary hold true. The difference is because the Process Handbook allows specialization by deletion of a decomposition, attribute or coordination mechanism; thus, for instance, some parts of a decomposition may be omitted in a specialized process of a parent process.

Figure 2 shows the relationship between specialization and decomposition in the Process Handbook methodology of process representation. The process “Sell product” is decomposed into the activities “Identify prospects”, “Inform prospects about product”, “Obtain order”, “Deliver product” and “Receive payment”. The specializations of “Sell product”, namely “Direct mail sales” and “Retail storefront sales” inherit the decompositions of “Sell product” but with several modifications. For example, in Figure 1, the activity “Identify prospects” in the decomposition of “Direct mail sales” has been replaced by “Obtain mailing lists”. When an inherited activity is replaced in a decomposition, it is necessary that the activity be replaced by a specialization of itself. Thus in Figure 1, “Obtain mailing lists” is a specialization of “Identify prospects". Also, since to run a retail storefront it is not necessary to identify prospects or obtain orders, these activities have been deleted in the inherited decomposition of “Retail storefront sales”.

---

2 Adapted from Malone, et. al (1993), Fig 1.
1 Note: This specialization relationship has not been explicitly shown in Figure 1.
Figure 2: Relationship between specializations and decompositions

Key

A

B

C

A is replaced by B in the specialization

A is omitted in the specialization

A must precede B

B and C are alternative specializations of A

B and C are decompositions of A

A is inherited in the specialization
2.2.2 Bundles

The schema for process representation also allows for combining a group of alternatively related specializations together into a bundle. For example, as shown in Figure 3, the bundles of “Sell Product” are “How sold?” and “What products?”, where the bundle “How sold?” consists of “Direct mail sales” and “Retail storefront sales”, while the bundle “What products?” consists of “Book sales” and “Clothing sales”. The Process Handbook methodology includes a tradeoff matrix that allows different alternatives within the same bundle to be compared in terms of specific goals. In addition, alternatives in a bundle automatically inherit alternatives from other bundles but not the other alternatives in their own bundle. For example, when selling clothes it is appropriate to be automatically provided with the alternatives for direct mail sales and retail sales, but not with the alternative of selling books.

![Diagram](image)

Figure 3: An example of bundles

2.2.3 Coordination

The Process Handbook uses the idea from coordination theory that coordination processes can be thought of as ways of managing dependencies among activities (Malone and Crowston, 1991). The Process Handbook methodology attempts to identify the various types of dependencies that are possible between activities and uses this concise
representation of coordination mechanism in the description of business processes. Examples of several common dependency types are listed in Table 1*.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Examples of coordination processes for managing dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Resource</td>
<td>“First come/first serve”, priority order, budgets, managerial decision, market-like bidding</td>
</tr>
<tr>
<td>Task Assignment</td>
<td>(same as for “Shared resources”)</td>
</tr>
<tr>
<td>Producer / consumer relationships</td>
<td></td>
</tr>
<tr>
<td>Prerequisite constraints</td>
<td>Notification, sequencing, tracking</td>
</tr>
<tr>
<td>Inventory</td>
<td>Inventory management (e.g., “Just In Time”, “Economic Order Quantity”)</td>
</tr>
<tr>
<td>Usability</td>
<td>Standardization, ask user, participatory design</td>
</tr>
<tr>
<td>Design for manufacturability</td>
<td>Concurrent design</td>
</tr>
<tr>
<td>Simultaneity constraints</td>
<td>Scheduling, synchronization</td>
</tr>
<tr>
<td>Task / subtask</td>
<td>Goal Selection, task decomposition</td>
</tr>
</tbody>
</table>

Table 1: Examples of common dependencies between activities and alternative coordination processes for managing them.

For instance, consider the prerequisite dependencies in Figure 2. There are no prerequisite dependencies in the generic process "Sell product", meaning that its decomposition of activities can be done in any particular order. However, in the more specialized processes "Direct mail sales" and "Retail storefront sales", there are explicit prerequisite dependencies between activities. For example, in "Direct mail sales", payment must be received before the product is delivered, whereas in a "Retail storefront sales", the shelves must be stocked before the customer pays at the register.

2.2.4 Context

When a modification is to be made to a decomposition, it is necessary to specify the context in which this modification is being made. For example, in Figure 2, consider the activity “Obtain order” that occurs in the decomposition of “Sell product” and is inherited in the decomposition of “Direct mail sales”. Suppose that it was desired to further

* Adapted from Malone, et. al (1993), Table 1. The list of dependency types in this table are not intended to be exhaustive.
decompose the activity “Obtain order”. This modification can occur in the context of “Direct mail sales”, or in the context of “Sell product”, or globally wherever the activity “Obtain order” is used. If the modification is made in the context of “Direct mail sales”, then the modification is visible only in the decomposition of “Direct mail sales”. If the modification is made in the context of “Sell product”, then the modification is visible in the decomposition of “Sell product” and all the specializations of “Sell product” that inherit the activity “Obtain order” in their decompositions, i.e., the activity “Direct mail sales”. If the modification is made globally, then the modification is visible in all decompositions that include the activity “Obtain order”.
3. System Overview

3.1 System Requirements

In implementing the on-line Process Handbook, it is necessary to first understand the specific requirements of such a system. These requirements are various functional specifications and practical considerations that need to be considered in the implementation of the Process Handbook. A list of these requirements is as follows\(^1\). It should be noted that there is a degree of overlap between the different categories.

3.1.1 Process Representation and Storage

One of the prime utilities of the Process Handbook system is that it serves as a repository for process descriptions. Applications that access data from this repository for the purposes of viewing, integrating, modifying, or entering process descriptions, need to be done using the handbook notions of specializations, decompositions and dependencies between activities. Thus, it is essential that any implementation of the system be able to encapsulate the handbook methodology of business process representation such that it allows for storage and retrieval of processes from the repository in terms of this methodology.

3.1.2 Inheritance

One of the key elements in the Process Handbook is its concise representation of processes using the notions of specialization and decomposition. As explained in Section 2.2.1, activities in the specialization hierarchy inherit attributes, decompositions and dependencies from their parents. It is necessary that any implementation of the system be able to support inheritance as used in the Process Handbook. Thus, whenever a new process is added to the specialization hierarchy, the system should allow for multiple

\(^1\) Primary investigation of requirements was done by Narasimha Rao, a former Research Assistant at Center for Coordination Science, MIT.
inheritance and automatically inherit the attributes, decompositions and dependencies from all its parents. Moreover, when a modification is made to an existing process in the specialization hierarchy, it is necessary to propagate the changes via inheritance throughout the system, such that future queries to the system will reflect consistent data.

3.1.3 Multiple Viewers/Editors

The system will have several viewers and editors to the underlying process descriptions in the repository and each of these viewers/editors will be able to represent the Process Handbook data in a format that is most suited to the task at hand. For instance, there will be a viewer/editor that graphically represents the specializations and decompositions of activities, there will be another viewer/editor that textually displays the specialization and decomposition hierarchies, and finally there will be a viewer/editor that allows for easy entry of semi-structured process descriptions. In addition to viewing processes in the repository in terms of the Process Handbook methodology, for the system to be useful, it should also be able to view and/or edit the data in other process formats, such as, IDEF0, data flow, etc. Thus, since the overall Process Handbook system will have multiple viewers/editors to the same data in the repository, it is important that any implementation of the system takes this fact into consideration.

3.1.4 Translation

Since the data in the repository can be viewed and edited in a variety of process formats, it is necessary for the system to be able to translate to and from different types of process formats into a common ontology. The format used for exchanging process representations in the Process Handbook is called the Process Interchange Format (PIF). The PIF format is an ontology for processes that is based upon the Knowledge Interchange Format (KIF) and is currently being developed by researchers at MIT, Stanford University, University of Toronto, and Digital Equipment Corporation (Lee, et al., 1994). To view the data in a process representation other than PIF or the Process Handbook representation, it will be usually necessary to first extract the data from the Process Handbook repository by
translating it to PIF. Once the data is in the PIF format, it can then be translated to the other process representation. Similarly, after a process description has been edited in a format other than PIF or the Process Handbook representation, it will usually be necessary to translate the process data from the other representation to PIF. Once the data is in the PIF format, it can then be translated to the Process Handbook representation which can be used to update the repository. Thus, it is important that any implementation of the system be able to translate data easily from the format of process representation in the repository (namely, the Process Handbook representation) to PIF and vice-versa.

3.1.5 Scaleable Design

The Process Handbook serves as a repository for business processes, and over time, the number of processes in the repository will continue to increase. Thus, it is important that any implementation of the system be scaleable such that it can handle a large number of business processes within the repository.

3.1.6 Practical Considerations

There are several practical considerations that must also be taken into account in the implementation of the Process Handbook. The system should be able to run on a PC or a Macintosh platform because these are the platforms that are most commonly available to end-users. Also, since one of the main uses of the Handbook is as a repository of processes, it is necessary that the physical storage of processes allows for multiple users to have access to this repository in a networked environment. Finally, the architecture of the system chosen should preferably require the least amount of development effort. This goal can be accomplished by using a componentware approach which involves taking advantage of the specific functionalities of different products and integrating them together in the overall Process Handbook system.
3.2 System Design: Three Tier Architecture

3.2.1 Description

Given the above system requirements, this thesis considers an implementation of the Process Handbook based on the three-tier client server architecture. The overall architecture for the Process Handbook consists of three components, as illustrated in Figure 4.

![Diagram of the Process Handbook architecture]

*Figure 4: The Process Handbook architecture*

The three components of the Process Handbook architecture are described as follows.

*Relational Database Management System (DBMS)*: This is the main storage component of the system and consists of a relational database, a database engine and the actual data file. The handbook representation for business processes has been modeled in the relational database using entity-relationship techniques. All process descriptions within the data file are stored in the relational database technology terms of records, tables and joins between tables. Since this component is simply static storage, it by itself does not have the ability to support the Process Handbook ideas of inheritance, but provides fields within tables that can be used by other components in the system to support inheritance.

*Process Handbook Logic Unit*: This component is a 'middleware' layer between the storage unit (i.e., the relational DBMS) and the viewers/editors. All accesses and modifications to the database from the viewers and editors occur through this middleware.
layer. This component provides an Application Programmers Interface (API) that abstracts the implementation of the storage device, namely the relational DBMS, and provides the viewers and editors with an API in terms of the Process Handbook methodology of process representation. Moreover, this component is also responsible for supporting the Process Handbook notions of inheritance. Whenever an editor modifies an existing process description, this component modifies the necessary records in the database tables for all affected processes according to the Process Handbook notions of inheritance.

Viewers/Editors: This component is the presentation logic and display unit and consists of all graphical user interfaces or applications that view and/or edit process descriptions from the underlying repository. All these viewers and editors access and modify the underlying database using only the API provided by the middleware “Process Handbook Logic Unit”. Thus the viewers/editors can access or modify data using the Process Handbook representation provided by the API and need not deal with the actual implementation of the underlying database or its inheritance mechanism. Some typical viewers/editors in the Process Handbook system include a graphical viewer/editor for specializations and decompositions, a text-based hierarchy viewer/editor, a data-flow viewer, a IDEF0 viewer, etc.

3.2.2 Advantages

The main advantage of the three-tier client server approach is that it decouples the Process Handbook logic from the viewers/editors and the relational DBMS. The middleware “Process Handbook Logic” layer consists of a set of functions that abstracts the implementation of the database from the viewers/editors and encapsulates the Process Handbook methodology of representing business process descriptions. This abstraction also masks the implementation of inheritance within the system from the viewers/editors. Thus the viewers/editors can be implemented using the middleware API in terms of the handbook representation and independent of the actual implementation of the relational database and the inheritance mechanisms. The fact that all viewers/editors share a
common middleware API has the advantage of consistency and easier maintenance because each viewer/editor need not implement its own data-processing engine to convert data from the table and record format in the relational database, to a format required by that particular viewer/editor.

Also, due to the abstraction mentioned above, the intermediary middleware insulates changes in the viewers/editors and database from each other and thus allows them to be modified independently. This fact allows for a componentware approach to building the Process Handbook system with incremental prototyping. Components can be added/changed to the viewers/editors and the DBMS with relative ease. In addition, the functionality provided by the middleware API can be incremented over time and thus allow for incremental prototyping.

Since the architecture is based on the client/server paradigm, the proposed architecture will also be scaleable. More computing power or storage space can be added to the client or server as needed such that the system can be easily scaled upward.

Finally, the proposed approach allows for flexibility because the middleware "Process Handbook Logic" unit can be placed physically, either on the client (viewer/editor), or on the server (relational DBMS) depending on the data processing characteristics. If the bulk of data processing occurs between the middleware and the client, then the middleware should be placed physically on the same machine as the client, whereas if the bulk of the data processing occurs between the database and the middleware, then the middleware should be placed physically on the server.

3.2.3 Disadvantages
One disadvantage to the three-tier client server architecture for the Process Handbook is system performance. Compared to a two-tier client server architecture, where the viewers/editors would be tightly coupled to the database implementation and would make accesses and modifications to the database in terms of the specific implementation of the
database, the three-tier architecture has an additional layer which makes processing slower. Certain viewers/editors which could make a smaller number of optimized queries to the database based on the specific database implementation, now need to go through the middleware layer and as a consequence, make a larger number of queries to the database and thereby degrade system performance.

### 3.3 Other Prototype Implementations

In addition to the three-tier client-server architecture outlined above, there have been other prototype implementations of the Process Handbook, each with its own task for which the implementation was well suited. In this section, each of these prototypes is described briefly.

#### 3.3.1 Kappa PC Implementation

The first Process Handbook implementation was developed using Intellicorp’s Kappa PC knowledge-based application building tool⁴. This prototype provided all the functionality and specifications of the Process Handbook methodology and was used to test the Process Handbook notions of specializations and decomposition. However, this implementation was limited in its ability to handle large number of processes and large amounts of unstructured text.

#### 3.3.2 Lotus Notes Implementation

The second Process Handbook implementation was developed using Lotus Notes⁷. This prototype was scaleable and allowed the entry of large amounts of semi-structured information about processes. However, this prototype did not have the capability to support the Process Handbook notions of inheritance amongst the activities in the

---

⁴ The Kappa PC prototype was developed primarily by Chris Dellarocas, a doctoral student at the Center for Coordination Science, MIT.

⁷ The Lotus Notes prototype was developed primarily by George Wyner, a doctoral student at the Center for Coordination Science, MIT.
specialization hierarchy. Moreover, this implementation was limited in its ability to graphically display process data.

3.3.3 Envision Implementation

The third Process Handbook implementation was developed using a CASE tool called Envision from Future Tech, Inc\textsuperscript{8}. This implementation allowed a graphical view of processes but did not have the capability to support the Process Handbook notions of inheritance amongst activities in the specialization hierarchy. Moreover, this tool was not suited for entering large amounts of semi-structured process descriptions.

\textsuperscript{8} The Envision prototype was developed primarily by Narasimha Rao, a former Research Assistant at Center for Coordination Science, MIT.
4. Database Design

4.1 Relational Database vs. Object-Oriented Database

As mentioned above, a relational database was used in the Process Handbook system architecture as the persistent store of process descriptions. Although an object-oriented database could also have been used for this purpose, it was decided to use a relational database because of the maturity of the technology and also because unlike an object-oriented database, the relational database model is well-defined. Moreover, not all the functionality of a truly object-oriented system was required.

In addition to the maturity of the relational technology, there were other motivating factors for choosing a relational database over an object-oriented one. A seemingly good reason to choose an object oriented database over a relational one would be inheritance; inheritance is a key component of an object-oriented database and would be already provided, whereas in a relational database, it would have to be modeled and implemented. However, there is a difference between data inheritance in an object-oriented database and inheritance in the Process Handbook that weakens the rationale for choosing an object-oriented database. In the true object-oriented sense, by inheritance, everything that is true for a parent is strictly true for its subclasses, effectively meaning that every instance of a subclass is also a member of its parent class. However, in the Process Handbook, everything that is true of a parent is not necessarily true of its children. This fact is because the Process Handbook allows specialization by a deletion of a decomposition, attribute or dependency. Thus, for instance, a specialized activity may inherit a set of decomposition activities from its parent, but may also delete one of the activities in the inherited decomposition. This fact is in contrast to a true-objected sense because everything that is true in the decomposition of the parent activity is not necessarily true in the decomposition of the specialized child activity.
Another reason for choosing a relational database over an object-oriented one is that most commercially available object-oriented databases lack the ability to add functionality to the application via an Application Programmers Interface (API). The ability to add functionality is critical in developing a system that would be able to represent processes using the Process Handbook methodology. Moreover, even if such an API were present, it would still require substantial programming effort to be able to represent processes using the handbook methodology. Also, most object-oriented systems cannot be integrated easily with other software components because the application and the database are tightly coupled together. For example, in developing an application on Intellicorp’s Kappa PC product, the final application and the database functionality are not separable because the application is closely linked to the underlying database.

Finally, as a practical consideration, there are no commercially available large scale object-oriented databases for PC’s or Macintoshes (which will be the primary platform for users), whereas there are many for relational databases.

4.2 Database Design

The relational database that was used to implement the Process Handbook was Microsoft Access. It should be noted that Microsoft Access was chosen for prototyping purposes, and because of the database independence provided by the middleware “Process Handbook Logic” unit (as discussed in Section 5.1.2), it can be easily changed to a more scaleable database, such as Oracle or Sybase.

The Process Handbook methodology for process representation was modeled into a relational database format using entity-relationship techniques. Although the relational model does not support inheritance by itself, the design provides fields within several tables that can be used by the middleware “Process Handbook Logic” unit to support

---

9 Primary modeling of the Process Handbook methodology in a relational database was done by Abraham Bernstein, a visiting Research Scientist at the Center for Coordination Science, MIT.
inheritance. The entity relationship diagram is as shown in Figure 5. It should be noted that a table suffixed with ".1" denotes the same table as the one without this suffix; tables with the suffix ".1" have been shown in the entity relationship diagram to denote relationships within the table itself. Thus, for example, "Attribute" and "Attribute_1" represent the same "Attribute" table within the relational database.
5. Middleware

As described in Section 3.2.1, the “Process Handbook Logic” unit is a middleware layer between the relational database and the viewers/editors. This middleware layer abstracts the implementation of the database from the viewers/editors and is the “logic” of the system, i.e., the place where the raw data specific to the database implementation gets converted into a format that is consistent with the Process Handbook representation.

The middleware layer has an API which provides the viewers and editors with a framework for data access and modification in terms of the Process Handbook methodology of process representation. Moreover, this middleware layer is also responsible for supporting the Process Handbook notions of inheritance. In implementing the overall architecture of the middleware, the middleware was separated into two modules - the middleware API module (called DB-Layer) and the inheritance module (called INH-Layer) as depicted in Figure 6.

![Figure 6: Relationship between API module and inheritance module](image)

The separation of the middleware into the two modules allows for an abstraction between the middleware API and the implementation of inheritance within the relational database. The functions within the inheritance layer should only be used internally by the functions within the middleware API and not by the viewers/editors. The viewers/editors make accesses and modifications to the Process Handbook repository only via the middleware.
API; if any of the these modifications affect inheritance within the repository, the middleware API then calls the appropriate functions within the inheritance layer to make the necessary modifications to the database tables.

**5.1 Middleware API**

**5.1.1 Description**

The middleware API is a set of functions used by the viewers/editors to access and modify data in the repository. This API provides the viewers and editors with a framework for data access and modification in terms of the Process Handbook methodology of process representation.

The middleware API was developed by incremental prototyping. This was done by developing the API in parallel with the development of the viewers and editors so that the desired functionality could be placed in the API. As more functions were needed by the viewers/editors to access and modify process descriptions, it was first determined whether that function belonged in the middleware API. If so, it was then determined how the function should be generalized in terms of the Process Handbook methodology such that it was not specific to the needs of one particular viewer/editor, but rather could be used by all other viewers/editors.

**5.1.2 Design & Implementation Considerations**

Since the bulk of the Process Handbook viewers/editors were being developed using Visual Basic, the middleware API was developed as set of Visual Basic functions. However, if need be, these functions can be ported as a C library to be used with other applications. To keep the number of functions within the middleware API at a manageable level, wherever possible, functions were parameterized either on the type of entity (i.e., activity, bundle, or dependency), or on the type of operation to be performed. The type arguments and return values of functions are all globally defined constants.
It was decided to make the middleware independent of the specific relational database (i.e., Microsoft Access) that was used in the initial implementation of the system. The rationale was that any relational database should be interchangeable as long as it had the same table design and the same relationships between tables. The advantage of being able to place any relational database is that, in the future it may be necessary to store a large amount of processes on a single database; and since Microsoft Access is not an extremely scaleable database, it would be advantageous if the database could be changed to some extremely scaleable database such as Oracle or Sybase and the rest of the system, including the middleware, left unchanged. This goal was achieved by using the Visual Basic programming language that allowed to make database independent Structured Query Language (SQL) queries and updates using Open Database Connectivity (ODBC).

The second main design consideration was to cache database tables to improve system performance. The rationale was that any viewer/editor would make many queries or updates to the database via the middleware API and thus, caching the database tables had the advantage that each query or update to the database would not have to go to disk if the records required from the tables were already cached in memory. This goal was achieved by using Visual Basic data-structures called dynasets. A Visual Basic dynaset can be a table, a relational join of tables, or the result of any query from the database. When the Process Handbook data file is first opened, a dynaset is created for all the necessary tables in the database. Visual Basic then internally handles the caching of the dynasets between memory and disk. Moreover, since the number of dynasets that can be concurrently used by Visual Basic is limited, having the necessary database tables shared amongst the various API functions provides for useful utilization of scarce resources.

Another main design consideration was the development of the notion of “focused” entities within the API. In the implementation of the middleware API, an entity in the Process Handbook is an activity, or bundle, or dependency. A focused entity can be thought of as a “viewpoint” to the process data in the repository, and at any one time,
there may be multiple “focused” entities. Related to each type of focused entity (i.e., activity, bundle, or dependency) there are several properties and methods that are accessible via the API. For example, the properties and methods of a focused activity include functions to get, set or delete its attributes, its first level of decomposition, its first level of specialization, and the dependencies in which the activity is involved. There are several advantages to be had from this notion of “focused” entities. First, the concept of “focused” entities closely parallels the anticipated patterns of usage by the viewers/editors. Second, all the necessary pointers and information can be aggregated together into a data-structure for that type of focused entity. This fact has the benefit that the user of the API need only provide the focused entity data-structure to various functions and need not be required to manage the various pointers and other information. Finally, the concept of “focused” entities provides an easy way to extend the API in a coherent fashion. For example, if a new type of entity is defined in the Process Handbook methodology and also in the implementation of the database, then the API can be easily extended by providing a new type of “focused” entity along with all its properties and methods in the middleware API.

Within the database implementation, activities and bundles are modeled within the same table, with the table having a bit field to distinguish between the two. Since the functionality provided by bundles and activities is nearly the same in the database implementation, the “focused” properties and methods of bundles and activities were aggregated together with the exception that the API does not allow a bundle to have a decomposition or take part in a dependency.

PIF is an integral part of the Process Handbook and is the format through which process descriptions are converted to and from non-Process Handbook representations. Several API functions were specifically geared towards providing the necessary functionality to allow process descriptions to be easily translated between the relational implementation of the Process Handbook storage and PIF.
Activities and dependencies can be inherited in the decomposition of an activity, and as described in Section 2.2.4, it is necessary to provide a context when a modification is being made. This context determines whether the proposed modification should occur wherever the activity or dependency was used, or only in the context of a particular activity’s decomposition and all its specializations. Thus, it was necessary to add a context flag argument to many of the API functions.

5.1.3 User Guide

Before any of the middleware API functions can be used, it is necessary to explicitly open the Process Handbook database using the “PHOpenDB” function. This function initializes the cached database tables and global variables. Also, after all API calls have been made by a viewer/editor it is necessary to explicitly close the database using the “PHCloseDB” function. This function ensures that all cached database tables are written back to the database.

The middleware API functions related to activities are shown in Table 2. Functions are categorized by what they affect and whether the function is an observer or a mutator.

<table>
<thead>
<tr>
<th>Activity Creation</th>
<th>Focused Activity Function?</th>
<th>Observers</th>
<th>Mutators</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td></td>
<td>PHNewActi</td>
</tr>
<tr>
<td>Activity Iterator</td>
<td>No</td>
<td>PHGetFirstEntity</td>
<td>PHGetNextEntity</td>
</tr>
<tr>
<td>Activity Name</td>
<td>No</td>
<td>PHGetName</td>
<td>PHChangeName</td>
</tr>
<tr>
<td>Activity Contact</td>
<td>No</td>
<td>PHGetContact</td>
<td>PHChangeContact</td>
</tr>
<tr>
<td>Activity Unique ID</td>
<td>No</td>
<td>PHGetUniqueID</td>
<td>PHChangeUniqueID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PHFindUniqueID</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decompositions, Where-Used, Specializations, Generalizations, Producer Activities, Consumer Activities, or LCA Activity of Activity</th>
<th>Focused Activity Function?</th>
<th>Observers</th>
<th>Mutators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>PHGetAll</td>
<td></td>
<td>PHAdd</td>
</tr>
<tr>
<td></td>
<td>PHGetFirst</td>
<td></td>
<td>PHDelete</td>
</tr>
<tr>
<td></td>
<td>PHGetNext</td>
<td></td>
<td>PHMoveSpec</td>
</tr>
<tr>
<td>Attributes of Activity</td>
<td>PHSetAttr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>PHGetAttr</td>
<td></td>
<td>PHDeleteAttr</td>
</tr>
</tbody>
</table>

Table 2: Middleware API functions related to activities
Before using functions that are labeled as focused activity functions, it is necessary to call the function “PHSetFocusActi”. This function modifies a “PHFocusActiObject” data-structure argument that is used in subsequent calls to focused activity functions. It should be noted that it is possible to have as many focused activities as needed. A complete specification of each of the middleware API functions is provided in Appendix A.

The middleware API functions related to bundles are shown in Table 3. Functions are categorized by what they affect and whether the function is an observer or a mutator. As discussed in Section 5.1.2, the focused functions for bundles and activities were aggregated together. Thus, before using functions that are labeled as focused bundle functions, it is necessary to call the function “PHSetFocusActi”. This function modifies a “PHFocusActiObject” data-structure argument that is used in subsequent calls to focused bundle functions. It should be noted that it is possible to have as many focused bundles as needed. A complete specification of each of the middleware API functions is provided in Appendix A.

<table>
<thead>
<tr>
<th>Bundle Creation</th>
<th>Focused Bundle Function?</th>
<th>Observers</th>
<th>Mutators</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td></td>
<td>PHNewBund</td>
</tr>
<tr>
<td>Bundle Iterator</td>
<td>No</td>
<td>PHGetName</td>
<td>PHChangeName</td>
</tr>
<tr>
<td>Bundle Name</td>
<td>No</td>
<td></td>
<td>PHChangeName</td>
</tr>
<tr>
<td>Bundle Contact</td>
<td>No</td>
<td>PHGetName</td>
<td>PHChangeContact</td>
</tr>
<tr>
<td>Bundle Unique ID</td>
<td>No</td>
<td>PHFindUniqueID</td>
<td>PHChangeUniqueID</td>
</tr>
<tr>
<td>Specializations or Generalizations of Bundle</td>
<td>Yes</td>
<td>PHGetAll</td>
<td>PHAdd</td>
</tr>
<tr>
<td>Attributes of Bundle</td>
<td>Yes</td>
<td>PHGetFirstAttr</td>
<td>PHDelete</td>
</tr>
</tbody>
</table>

| PHGetFirstEntity |
| PHGetNextEntity |
| PHGetName |
| PHGetNext |
| PHGetUniqueID |
| PHFindUniqueID |
| PHValueAttr |

| PHGetFirstAttr |
| PHGetNextAttr |
| PHSetAttr |
| PHDeleteAttr |

Table 3: Middleware API functions related to bundles

The middleware API functions related to dependencies are shown in Table 4. Functions are categorized by what they affect and whether the function is an observer or a mutator.
Before using functions that are labeled as focused dependency functions, it is necessary to call the function “PHSetFocusDepn”. This function modifies a “PHFocusDepnObject” data-structure argument that is used in subsequent calls to focused dependency functions. It should be noted that it is possible to have as many focused dependencies as needed. A complete specification of each of the middleware API functions is provided in Appendix A.

<table>
<thead>
<tr>
<th>Dependency Type</th>
<th>Focused Dependency Function?</th>
<th>Observers</th>
<th>Mutators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependency Creation</td>
<td>No</td>
<td></td>
<td>PHNewDepn</td>
</tr>
<tr>
<td>Dependency Iterator</td>
<td>No</td>
<td>PHGetFirstEntity PHGetNextEntity</td>
<td></td>
</tr>
<tr>
<td>Dependency Name</td>
<td>No</td>
<td>PHGetName</td>
<td>PHChangeName</td>
</tr>
<tr>
<td>Dependency Contact</td>
<td>No</td>
<td>PHGetContact PHFindUniqueID</td>
<td>PHChangeUniqueID</td>
</tr>
<tr>
<td>Dependency Unique ID</td>
<td>No</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ancestor Activity of Dependency</td>
<td>Yes</td>
<td>PHGetAcsActi</td>
<td>PHChangeAcsActi</td>
</tr>
<tr>
<td>Producer Activities of Dependency</td>
<td>Yes</td>
<td>PHGetFirstProdActi PHGetNextProdActi</td>
<td>PHAddProdActi PHDeleteProdActi</td>
</tr>
<tr>
<td>Consumer Activities of Dependency</td>
<td>Yes</td>
<td>PHGetFirstConsActi PHGetNextConsActi</td>
<td>PHAddConsActi PHDeleteConsActi</td>
</tr>
<tr>
<td>Attributes of Dependency</td>
<td>Yes</td>
<td>PHGetFirstDAt PHGetNextDAt PHFindDAt PHValueDAt</td>
<td>PHSetDAt PHDeleteDAt</td>
</tr>
</tbody>
</table>

*Table 4: Middleware API functions related to dependencies*

### 5.2 Inheritance

As described in Section 5, the inheritance mechanism has been placed in a separate module from the middleware API functions that are visible to the viewers/editors. The API functions call the appropriate functions within the inheritance layer to make the necessary inheritance modifications to the database tables.

In the Process Handbook and the middleware, inheritance occurs through the specialization hierarchy of activities. The inheritance mechanism supported by the
middleware consists of the inheritance of attributes, the inheritance of decomposition activities and the inheritance of dependencies. Every activity in the specialization hierarchy inherits the attributes and decompositions including dependencies of all its parents. In the following sections, each of the inheritance mechanisms is described in detail, and then examples of inheritance, as modeled within the middleware, are provided\textsuperscript{10}.

5.2.1 Attribute Inheritance

5.2.1.1 Description

Attribute inheritance in the Process Handbook is similar to the inheritance of attributes as used in the object-oriented sense of computer science. If each attribute is considered as a name-value pair, then each activity in the specialization hierarchy should inherit the name-value pairs of all its parents and should allow for the modification of the values of the inherited name-value pairs. Whenever a change (i.e., modification or deletion) is made to a name-value attribute pair, that change should also be reflected in all the name-value pairs which inherit from this particular name-value attribute pair. For example, if the value is modified in a name-value pair, the values of all attributes that inherit the value of this name-value attribute should also be modified.

Since the specialization hierarchy allows for multiple inheritance, i.e., an activity can have more than one parent, there is the possibility of name clashes for inherited attributes. Ideally, in the object-oriented sense, this problem can be solved by inheriting each name-value pair separately and then providing a way to distinguish between them in new methods (Kim & Lochovsky, 1989). However, in the implementation of the system, a more naive approach is taken which is described in the following section on design and implementation considerations.

\textsuperscript{10} The modeling of inheritance within the middleware was initially begun by Abraham Bernstein, a visiting Research Scientist at the Center for Coordination Science, but was subsequently taken over by the author.
5.2.1.2 Design and Implementation Considerations

One of the main design considerations in implementing the inheritance of attributes within the relational schema was whether inherited attribute values should be flattened or non-flattened. By flattened inheritance of attributes, there would be a record in the Attribute table for every attribute of every activity, including inherited attributes. By non-flattened inheritance, there would be a record in the Attribute table for only non-inherited attributes of every activity. To find an attribute value with the flattened inheritance implementation, it is only necessary to look at the appropriate attribute record for that activity. However, in the non-flattened inheritance implementation, it may be necessary to look at more than one record. First it is checked whether the attribute for that activity exists in the attribute table. If it is not present, it is required to look whether the attribute record exists in any of its parents in the specialization hierarchy. In this fashion, it is necessary to look at all the ancestors of the activity until the nearest ancestor with that attribute is found.

In the implementation of inheritance within the database, a flattened approach was taken. There were several tradeoffs that were considered in choosing to implement flattened inheritance over non-flattened inheritance. First, although the attribute table size with flattened inheritance would be significantly larger than in the non-flattened case, this implementation will usually provide for faster queries of attribute values. This fact is not self-evident because queries in the flattened implementation have to search through a significantly larger attribute table, whereas attribute queries in the non-flattened case may require several queries to a smaller attribute table. However, under the assumption that the attributes to be queried will on average be several levels deep in terms of where the attribute is inherited from (thereby requiring several queries in the non-flattened implementation), and that in the flattened implementation, the attribute table has appropriate indexes, then on average, attribute queries will be faster in the flattened implementation. Although flattened inheritance provides, on average, for faster attribute queries, it is slower for making modifications to attributes. This slower modification time is due to the fact that by flattened inheritance, a modification or creation of a new
attribute record means that a modification or creation of a new attribute record must also occur for each of the descendants of the original activity, whereas by non-flattened inheritance, the modification or creation of a new attribute record need not be propagated. The rationale for choosing the flattened implementation of inheritance rather than the non-flattened inheritance with the Process Handbook system was that under the typical usage scenario, there are likely to be a significantly larger number of attribute value queries as opposed to attribute modifications.

One side benefit of the flattened implementation of inheritance is that it allows an inherited attribute to be deleted from an activity in the specialization hierarchy without affecting the attribute in its parents. Thus it is possible for an attribute to be present in a parent activity but not in its descendants. This fact is because by the flattened implementation of inheritance we can easily delete the appropriate attribute records. This is in contrast to a non-flattened implementation where the deletion of an inherited attribute would have to be modeled by setting the value of the inherited attribute slot to “Deleted”, as opposed to deleting the slot itself. In this case, the values of attributes with value “Deleted” would need to be suppressed.

The flattened inheritance was modeled within the attribute table with the aid of the two fields atri_inherit_from_atri_id and atri_inherit_value_from_atri_id. The former field (henceforth called the slot-pointer) is a pointer to the attribute table record of the highest ancestor from which this attribute name slot is inherited. The latter field (henceforth called the value-pointer) is a pointer to the attribute table record of the highest ancestor from which this attribute value is inherited. For instance, consider the specialization hierarchy in Figure 7 annotated with the slot-pointer and value-pointer for the attribute “Cost”.

For the “Cost” attribute of the activity “Mail order sales”, both the slot and the value pointers point to itself because neither the attribute slot nor the attribute value is inherited. For the “Cost” attribute of “Mail order clothing sales”, the slot is inherited and the slot
pointer points to the “Cost” attribute of “Mail order sales”. However, since the value of the “Cost” slot has been changed locally to “Mail order sales”, the value-pointer points to itself. For the “Cost” attribute of “L. L. Bean”, the slot is inherited and points to the highest ancestor from which this slot is inherited, namely the “Cost” attribute of “Mail order sales”. Finally since the value of the “Cost” slot is also inherited, the value-pointer points to the highest ancestor from which the value is inherited, namely “Mail order clothing sales”.

The implementation of inheritance within the system allows for multiple inheritance. Each activity inherits all the attributes of all its parents. However, when there is a name-clash (i.e., the same slot inherited from different parents with differing values), the implementation is somewhat naive. The system automatically takes the value of the first parent it encounters and ignores the other value(s). This behavior is not ideal, and in the future, the system should be augmented to either prompt the user to ask which value to
inherit, or it should store both values and allow viewers/editors to choose the appropriate attribute value.

5.2.1.3 Example

To illustrate the mechanism of attribute inheritance as modeled within the middleware, consider the specialization hierarchy of “Sell Product” as shown in Figure 8.

![Diagram](image)

**Figure 8: Example specializations of “Sell Product”**

Suppose the attribute “Cost” is added to “Sell product” and its value is set to “high”. Then by attribute inheritance, all other activities that are specializations of “Sell product” will inherit this attribute slot and value. To illustrate this attribute inheritance, consider the activities “Lands End” and “L. L. Bean” and all their ancestors in the specialization
hierarchy. The values of the “Cost” attribute for each of these ancestors are as tabulated in the column “Step 1” of Table 5.

<table>
<thead>
<tr>
<th></th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sell product</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Mail order sales</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>DELETED</td>
</tr>
<tr>
<td>Clothing sales</td>
<td>high</td>
<td>high</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>Mail order clothing sales</td>
<td>high</td>
<td>low</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>Lands End</td>
<td>high</td>
<td>low</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>L. L. Bean</td>
<td>high</td>
<td>low</td>
<td>low</td>
<td>low</td>
</tr>
</tbody>
</table>

*Table 5: Value of attribute "Cost" at subsequent steps*

Now suppose that the value of the attribute “Cost” if changed in “Mail order clothing sales” to “low”. Then by inheritance, the values of the “Cost” attribute for “Lands End” and “L. L. Bean” will also be changed to “low”. The new values of the “Cost” attribute will now be as tabulated in the column “Step 2” of Table 5. As the next step, suppose that the value of the “Cost” attribute is changed in the activity “Clothing sales” to “medium”. Then the “Cost” attribute values for “Mail order clothing sales”, “Lands End”, and “L. L. Bean” will remain unchanged as “low”. This fact is because the value of the “Cost” attribute is not inherited in “Mail order clothing sales” and its value is the value that is inherited by “Lands End” and “L. L. Bean”. The new values of the “Cost” attribute will now be as tabulated in the column “Step 3” of Table 5. As a final step, suppose that the “Cost” attribute is deleted for “Mail order business” 11. The deletion of the “Cost” attribute is not propagated by inheritance to “Mail order clothing sales” and its specializations. This fact is because the “Cost” attribute in “Mail order clothing sales” was multiply inherited from “Mail order business” and “Clothing sales”. If the activity “Mail order clothing sales” had not been multiply inherited, then the “Cost” attributes of “Mail order clothing sales”, “Lands End”, and “L. L. Bean” would also have been deleted by inheritance. The new values of the “Cost” attribute are as tabulated in the column “Step 4” of Table 5.

11 As discussed in Section 5.2.1.2, the flattened implementation of attribute inheritance allows inherited attribute slots to be deleted. This is in contrast to a non-flattened implementation where the deletion would have to be modeled by setting the value of the inherited attribute to “Deleted”, as opposed to deleting the slot itself.
5.2.2 Decomposition Inheritance

5.2.2.1 Description

In the Process Handbook, the inheritance of decompositions occurs via the specialization hierarchy. Each activity in the specialization hierarchy inherits the decompositions of all its parents. These inherited decompositions can be augmented at a specialized level by the addition, replacement, or deletion of decomposition activities or dependencies. Whenever a modification is made to a decomposition, that change should be reflected in all inherited decompositions (provided that the decomposition activity or dependency that is being modified has not been deleted in the decomposition of the specialized activity).

The inheritance of decompositions in the case of multiple inheritance is not obvious. For example, suppose that the decomposition of one parent was “Obtain order” and “Deliver product” and the decomposition of another parent was “Obtain order” and “Stock shelves”. The simplest approach to multiple inheritance would be to take the union of all the decompositions of an activity’s parents allowing for duplicate decompositions between the activity and one of its decomposed activities. Using this strategy, the resultant inherited decomposition in the above example would be “Obtain order”, “Obtain order”, “Stock shelves” and “Deliver product”. An obvious improvement to this approach would be to check for duplicate decompositions and remove all but one of them from the inherited decomposition. Using this strategy, the resultant inherited decomposition in the above example would be “Obtain order”, “Stock shelves” and “Deliver product”. As a final level of sophistication, it would be appropriate to also remove decomposition activities that are further specialized in the decomposition of another parent activity. Using this strategy, since “Stock shelves” is a specialization of “Deliver product”, the resultant decomposition in the above example would be “Obtain order” and “Stock shelves”. In the current implementation of the multiple inheritance of decompositions within the middleware, checks are made only for duplicate decompositions between activities and not for specialized decomposition activities.
5.2.2.2 Design and Implementation Considerations

The main design consideration in implementing the inheritance of decompositions within the relational schema was whether inherited decompositions should be flattened or non-flattened within the database tables. The approach that was taken in the implementation of the system can be described as top-level-flattened because it is a mix between flattened and non-flattened. This implementation approach can be explained by Figure 9.

![Figure 9: Implementation of decomposition inheritance](image)

As depicted in Figure 9, the decompositions of “Run business unit” are “Acquire raw materials”, “Develop products” and “Sell product”. Moreover, the activity “Sell product” is further decomposed into “Obtain order”, “Deliver product” and “Receive payment”. When the activity “Retail storefront business” is added as a specialization of “Run business unit”, it inherits the decompositions of “Run business unit”. In the top-level-flattened implementation of the inheritance of decompositions, only the first level of inherited decompositions (as shown by the dotted lines in Figure 9) are flattened in the decomposition table. Thus, an inherited decomposition record is added in the decomposition table for all the decompositions of “Retail storefront sales” which are
shown by dotted lines. The decompositions of "Sell product" in "Retail storefront sales" are not added as inherited decompositions in the decomposition table. This fact is because the activity "Sell product" in the decomposition of "Retail storefront sales" is the exact same activity as in the decomposition of "Run business unit", and this activity already has the necessary non-inherited decompositions.

The top-level-flattened inheritance was modeled within the decomposition table with the aid of the field deco_inherit_from_deco_id. This field is a pointer to the decomposition record of the highest ancestor of the parent of the decomposition link from which this decomposition is inherited. This field is analogous to the slot-pointer field for attribute inheritance as described in Section 5.2.1.2 and serves the same purpose.

As described in Section 2.2.4, when a modification is made to an activity or dependency within a decomposition, it is necessary to specify the context in which this modification is being made. This context specifies whether the modification is to be made only in the decomposition of an activity and all its specializations, or whether the modification is to be made globally in all decompositions in the repository where the activity occurs. The implementation of context handling within the middleware is as follows. When the context is global, the modification is simply made to the affected activity or dependency without intervention by the middleware inheritance module. When a modification is being made to an activity or dependency in the context of a specific decomposition, a list of non-inherited decomposition ancestors for that activity/dependency is provided. Then in the specific context decomposition, each activity in the list is replaced by a specialization of itself. Moreover, by the inheritance of decompositions, this change is also propagated to all specializations that inherit from the specific context decomposition. Finally, the required modification is then made to the newly specialized activity or dependency. Thus, the mechanism described allows modifications in a decomposition to occur only in the context of specific decomposition. An example of the context mechanism is provided in the next section.
5.2.2.3 Example

To illustrate the context mechanism described above, consider the decomposition of "Run business unit" as shown in Figure 9. Suppose that "Retail storefront business" is added as a specialization of "Run business unit", then it inherits the decomposition of "Run business unit" as shown in Figure 9. Now suppose that it is desired to change the decomposition of "Sell product" only in the context of "Retail storefront sales" by replacing "Receive payment" with "Customer pays at register". The context mechanism works by first creating a new specialization of "Sell product" also called "Sell product". Then, in the decomposition of "Retail storefront business" the original "Sell product" is replaced by the specialized "Sell product". The specialized "Sell product" inherits the decompositions of the original "Sell product". As a final step, in the decomposition the specialized "Sell product", the activity "Receive payment" is replaced by "Customer pays at register". The final decompositions of "Run business unit" and "Run storefront business" are as depicted in Figure 10.

![Diagram of context mechanism]

Key

- A
- B
- C

B and C are decompositions of A, with C being inherited

Figure 10: Example of context mechanism

5.2.3 Dependency Inheritance

5.2.3.1 Description

As modeled within the database implementation and the middleware, a dependency consists of a set of producer and consumer activities with all such producer and consumer
activities belonging to a particular decomposition. The current implementation of the system cannot handle composite dependencies or dependencies which do not involve resources (i.e., producer and consumer activities). Associated with each dependency is a least common ancestor (LCA) activity, which is an activity that is the closest ancestor in the decomposition hierarchy to all the producer and consumer activities. The LCA activity can be thought of the activity to which the dependency belongs. The inheritance of dependencies occurs via the LCA activity.

When a new dependency is added between a set of producer and consumer activities within a decomposition hierarchy, the inheritance of dependencies operates as follows. A new dependency is created (if the same dependency does not already exist) for each of the specializations of the LCA activity. For each of these new dependencies, the LCA (henceforth called new-LCA) is set to the specialized activity of the original-LCA activity. The set of producer and consumer activities for each of these new dependencies consists of the same set of producers and consumers as in the original dependency (provided they all occur in the decomposition of the new-LCA activity), or any set which consists of the specialization of these activities (provided they all occur in the decomposition of the new-LCA activity).

When a dependency is deleted from a decomposition, the dependency is also deleted from all the decompositions of the specializations of the LCA activity that had previously inherited the dependency in the method outlined above. Moreover, when a dependency is modified (for example, a new producer or consumer is added or deleted), this modification should be inherited in all the decompositions of the specializations of the LCA activity that had previously inherited the dependency in the method outlined above.

In the implementation of the inheritance in the middleware, the addition and deletion of dependencies has been appropriately modeled. However, the current implementation of the system does not propagate inheritance modifications of dependencies. The work
around in the current system is to delete the old dependency and add a new dependency with the necessary modifications.

5.2.3.2 Design and Implementation Considerations

In the implementation of the inheritance of dependencies within the relational schema, the choice between a flattened implementation and a non-flattened implementation was relatively easy. This fact is because although inherited dependencies may relate the same set of producer and consumer activities, these dependencies are in different decompositions and have different LCA activities. Thus, since inherited dependencies have different LCA activities, it was decided to implement the inheritance of dependencies via flattened records in the relational database tables.

The flattened inheritance was modeled within the dependency table with the aid of the field depe_inherit_from_depe_id. This field is a pointer to the dependency record of the highest ancestor of the LCA activity from which this dependency is inherited. This field is analogous to the slot-pointer field for attribute inheritance as described in Section 5.2.1.2 and serves the same purpose.

Also as discussed above, the current implementation of the inheritance of dependencies only propagates dependency additions and deletions, but not modifications.

5.2.3.3 Example

To illustrate the mechanism of dependency inheritance as modeled within the middleware, consider the decompositions of the activities “Sell product”, “Mail order sales” and “Mail order clothing sales” as shown in Figure 11. “Mail order sales” is a specialization of “Sell product” and inherits its decomposition. Also, “Mail order clothing sales” is a specialization of “Mail order sales” and inherits its decompositions, with the exception that “Deliver product” is replaced by the specialized activity “Mail clothes”. The inherited decompositions are denoted by dashed lines in Figure 11.
Figure 11: Example decompositions before addition of dependency

Now, suppose that a pre-requisite dependency is to be added from “Receive payment” to “Deliver product” in the context of “Mail order sales”. By the context mechanism described in 5.2.2.2, in the decomposition of “Mail order clothing sales”, the activities “Receive payment” and “Deliver product” which were inherited, will be replaced by specializations of themselves (henceforth called specialized “Receive payment” and specialized “Deliver Product”). Moreover by the inheritance of decompositions, the “Receive payment” activity in the decomposition of “Mail order clothing sales” is replaced by the specialized “Receive payment” that was created by the context mechanism. The prerequisite dependency is then added between the specialized “Receive payment” and the specialized “Deliver product” in the decomposition of “Mail order sales”. By the inheritance of dependencies, a new dependency is also added between the specialized “Receive payment” and “Mail clothes” in the decomposition of “Mail order clothing sales” because “Mail clothes” is a specialization of “Deliver Product”. The resultant decompositions of “Sell Product”, “Mail order sales” and “Mail order clothing sales” are as follows:

- Sell product
  - Obtain Order
  - Receive payment
  - Deliver product

- Mail order sales
  - Obtain Order
  - Receive payment
  - Deliver product

- Mail order clothing sales
  - Obtain Order
  - Receive payment
  - Mail clothes

Key:

A

B and C are decompositions of A, with C being inherited
sales” is as shown in Figure 12, with inherited decompositions denoted by dashed lines. It should be noted that Figure 12 does not explicitly represent the fact that the activities “Receive payment” and “Deliver product” in the decompositions of “Mail order sales” and “Mail order clothing sales” are actually specializations of the activities with the same name in the decomposition of “Sell product”. Also, Figure 12 does not explicitly represent the fact that the activity “Obtain order” in the decompositions of “Mail order sales” and “Mail order clothing sales” is actually the same activity as in the decomposition of “Sell product”.

Figure 12: Example decompositions after addition of dependency
6. Future Work

There are several modifications that can be made to the middleware data abstraction to increase its functionality.

The first modification is the handling of multiple inheritance with regard to attributes and decompositions. As discussed in Section 5.2.1.2, when a name clash occurs in attribute inheritance, the middleware inheritance module automatically takes the value of the first parent it encounters and ignores the values of its other parents. This behavior is not ideal and in the future, the middleware inheritance module should be augmented to prompt the user to ask which value of its multiple parents to inherit, or it should store all values and allow viewers/editors to choose the appropriate attribute value. Also, as discussed in Section 5.2.2.1, when inheriting multiple decompositions, the middleware inheritance module automatically takes the union of all the decompositions of all its parents, with the removal of duplicate decompositions between an activity and its decomposed activities. It would be appropriate if the middleware inheritance module was augmented such that when inheriting multiple decompositions, if multiple parents have the same decomposition activity and this activity is specialized in one of the parents, then only the specialized activity should be inherited in the decomposition.

There are several modifications which could be made with regard to dependencies. First, a dependency is currently modeled by a set of producer and consumer activities within the relational schema of the database. As a result, the middleware data abstraction cannot handle composite dependencies or dependencies which do not involve resources. When the relational schema of the database has been modified to handle this deficiency, the middleware API and the inheritance modules need to be extended. Also, currently there is a disconnect between some of the middleware API functions related to dependencies and the inheritance layer. After a dependency has been added with all its producer and consumer activities and the least common ancestor has been set, it is necessary for the viewer/editor to call explicitly a function within the inheritance layer to propagate the
dependency. Moreover, when a dependency is to be added/deleted in the context of a specific decomposition, it is necessary for the viewer/editor to call explicitly a function within the inheritance layer to enable the context mechanism. Ideally, the viewers and editors should make accesses and modifications to the Process Handbook repository only via the middleware API and not directly through the inheritance layer; if any of the modifications affect inheritance within the repository, the middleware API should call the appropriate function in the inheritance layer. Thus, it would be suitable to modify the appropriate functions in the middleware API related to dependencies such that there is no disconnect between these functions and the inheritance layer. Finally, as discussed in 5.2.3.1, the implementation of inheritance of dependencies only propagates dependency additions and deletions and does not propagate dependency modifications. The current work around to propagate dependency modifications is to delete the affected dependency and add a new dependency with the necessary modifications. It would be appropriate if the data abstraction was modified to deal explicitly with inheritance propagation on dependency modification.
7. Conclusion

In the implementation of the Process Handbook system, a three tier client-server architecture was employed in which the first tier was a relational database, the second tier was a “Process Handbook Logic” layer, and the third tier was applications that view/edit data from the repository. This thesis considered the implementation of the middle “Process Handbook Logic” tier as a data abstraction. This data abstraction provided a middleware API that was used by the viewers/editors to access data from the repository in terms of the Process Handbook methodology of business process representation and independent of the actual implementation of the storage device, i.e., the relational database. Moreover, the data abstraction was responsible for supporting the Process Handbook notions of inheritance. Viewers/editors were transparent to the fact that the inheritance mechanism was being provided on top of a relational database.

The data abstraction with inheritance that this thesis implemented has been used in the development of several viewers/editors. These viewers and editors have served as test cases to verify the validity and usability of the data abstraction. The main viewer/editor that tested the functionality of data abstraction was the PIF Toolkit (Chan, 1995). In addition, other viewers that were developed using the data abstraction include an IDEF0 viewer\textsuperscript{12}, a data flow viewer\textsuperscript{13} and an outline viewer\textsuperscript{14}.

The implementation of the above mentioned viewers/editors based on the data abstraction has also verified many of the advantages that were initially sought. It has definitely allowed for the implementation of the viewers/editors in terms of the Handbook methodology and independent of the implementation of the storage device, i.e., the

\textsuperscript{12} The IDEF0 viewer was developed by Narasimha Rao, a former Research Assistant at Center for Coordination Science, MIT.

\textsuperscript{13} The data flow viewer was developed by Naved Khan, a Research Assistant at Center for Coordination Science, MIT.

\textsuperscript{14} The outline viewer was developed jointly by the author, Frank Chan and Naved Khan. All implementers are Research Assistants at the Center for Coordination Science, MIT.
relational database. Moreover the data abstraction had the advantage of consistency and easier maintenance because each viewer/editor did not need to implement its own data-processing engine to convert data from the table and record format in the relational database to a format required by that particular viewer/editor. Finally, the data abstraction insulated changes between the viewers/editors and the database. This fact allowed for a componentware approach to building the overall Process Handbook system because the specific functionalities of different products were taken advantage of and integrated together in the overall Process Handbook system.
References


Appendix A: Middleware API Specifications

An alphabetized listing of the middleware API specifications is provided below.

Name : PHAdd
Syntax : PHAdd (next_type As Integer, my_object As PHFocusActiObject, entity_id As Long, context_ids() As Long, new_acti_ids() As Long)
Returns : Long
Requires : When a decomposition or where-used link is added, context_ids is a bottom up array of activity ids representing the context in which the addition is to be made.

context_ids and new_acti_ids are different arrays, unless there is no context, in which case both context_ids and new_acti_ids should be set to the global constant PH_NO_CONTEXT()
Modifies : my_object, new_acti_ids
Effects : Adds a new relationship defined by the type to the focused object. The valid next_types are as follows:

PH_DECOMPOSITION
PH_WHERE_USED
PH_SPECIALIZATION
PH_GENERALIZATION
PH_CONSUMER_DEPENDENCY
PH_PRODUCER_DEPENDENCY
PH_ANCESTOR_DEPENDENCY
PH_UNDEFINED_TYPE

new_acti_ids is modified to be the bottom up list of newly specialized activity ids created by the context mechanism.

Returns PH_SUCCESS if all is O.K.
Returns PH_DUPLICATE if the relationship already exists
Returns PH_ERROR if an error is encountered
Returns PH_WRONG_TYPE if the next_type is an undefined type
Returns PH_NOT_ACTIVITY if a decomposition or where_used is added to a currently focused bundle

Name : PHAddConsActi
Syntax : PHAddConsActi (my_object As PHFocusDepnObject, activity_id As Long)
Returns : Long
Requires :
Modifies :
Effects : Adds a new consumer activity to the focused dependency

Returns PH_SUCCESS if all is O.K.
Returns PH_DUPLICATE if the relationship already exists
Returns PH_ERROR if an error is encountered

NOTE: To propagate the inheritance of dependencies, it is necessary to first set all producer and consumer activities, set the least common ancestor and then explicitly call the function Dependency_Added from the inheritance layer

63
Name : PHAddProdActi
Syntax : PHAddProdActi (my_object As PHFocusDepnObject, activity_id As Long)
Returns : Long
Requires : 
Modifies : 
Effects : Adds a new producer activity to the focused dependency

Returns PH_SUCCESS if all is O.K.
Returns PH_DUPLICATE if the relationship already exists
Returns PH_ERROR if an error is encountered

NOTE: To propagate the inheritance of dependencies, it is necessary to first set all producer and consumer activities, set the least common ancestor and then explicitly call the function
Dependency_Added from the inheritance layer

Name : PHChangeAucsActi
Syntax : PHChangeAucsActi (my_object As PHFocusDepnObject, activity_id As Long)
Returns : Long
Requires : 
Modifies : 
Effects : Changes the least common ancestor of the focused dependency to the specified activity_id.

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the dependency cannot be found
Returns PH_NOT_ACTIVITY if the specified activity_id is a bundle
Returns PH_ERROR if an error is encountered

NOTE: To propagate the inheritance of dependencies, it is necessary to first set all producer and consumer activities, set the least common ancestor and then explicitly call the function
Dependency_Added from the inheritance layer

Name : PHChangeContact
Syntax : PHChangeContact (entity_type As Integer, entity_id As Long, entity_contact As String) As Long
Returns : Long
Requires : 
Modifies : 
Effects : Changes the contact of the specified entity_id. Valid entity_types are as follows:

PH_ACTIVITY
PH_BUNDLE
PH_DEPENDENCY

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the ID cannot be found
Returns PH_ERROR if an error is encountered
Name : PHChangeName
Syntax : PHChangeName (entity_type As Integer, entity_id As Long, entity_name As String)
Returns : Long
Requires :
Modifies :
Effects : Changes the name of the specified entity_id. Valid entity_types are as follows:
PH_ACTIVITY
PH_BUNDLE
PH_DEPENDENCY

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the ID cannot be found
Returns PH_ERROR if an error is encountered

Name : PHChangeUniqueID
Syntax : PHChangeUniqueID (entity_type As Integer, entity_id As Long, entity_unique_id As String)
Returns : Long
Requires :
Modifies :
Effects : Changes the unique id of the specified entity_id. Valid entity_types are as follows:
PH_ACTIVITY
PH_BUNDLE
PH_DEPENDENCY

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the ID cannot be found
Returns PH_ERROR if an error is encountered

Name : PHCloseDB
Syntax : PHCloseDB()
Returns : Long
Requires : A database has previously been opened using the PHOpenDB function and is still open.
Modifies :
Effects : Closes all the dynasets of the database, and then closes the database itself.

Returns PH_SUCCESS if all is O.K.
Returns PH_ERROR if an error is encountered
Name : PHDelete
Syntax : PHDelete (next_type As Integer, my_object As PHFocusActiObject, entity_id As Long, context_ids() As Long, new_acti_ids() As Long)
Returns : Long
Requires : When a decomposition or where-used link is deleted, context_ids is a bottom up array of activity ids representing the context in which the deletion is to be made.

context_ids and new_acti_ids are different arrays, unless there is no context, in which case both context_ids and new_acti_ids should be set to the global constant PH_NO_CONTEXT()
Modifies : my_object, new_acti_ids
Effects : Deletes the relationship defined by the type to the focused object. The valid next_types are as follows:

PH_DECOMPOSITION
PH_WHERE_USED
PH_SPECIALIZATION
PH_GENERALIZATION
PH_CONSUMER_DEPENDENCY
PH_PRODUCER_DEPENDENCY
PH_UNDEFINED_TYPE

new_acti_ids is modified to be the bottom up list of newly specialized activity ids created by the context mechanism.

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the relationship does not exist
Returns PH_ERROR if an error is encountered
Returns PH_WRONG_TYPE if the next_type is invalid
Returns PH_NOT_ACTIVITY if a decomposition or where_used is deleted to a currently focused bundle

Name : PHDeleteAttr
Syntax : PHDeleteAttr (my_object As PHFocusActiObject, attr_name As String, context_ids() As Long, new_acti_ids() As Long)
Returns : Long
Requires : When the attribute is to be deleted in the context of a specific decomposition, context_ids is a bottom up array of activity ids representing the context in which the deletion is to be made.

context_ids and new_acti_ids are different arrays, unless there is no context, in which case both context_ids and new_acti_ids should be set to the global constant PH_NO_CONTEXT()
Modifies : new_acti_ids
Effects : Deletes the specified attribute of the focused activity

new_acti_ids is modified to be the bottom up list of newly specialized activity ids created by the context mechanism.

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the attribute does not exist
Returns PH_ERROR if an error is encountered
Name : PHDeleteConsActi
Syntax : PHDeleteConsActi (my_object As PHFocusDepnObject, activity_id As Long)
Returns : Long
Requires :
Modifies :
Effects : Deletes the consumer activity id from the focused dependency object.

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the relationship does not exist
Returns PH_ERROR if an error is encountered

Name : PHDeleteDAtr
Syntax : PHDeleteDAtr (attr_name As String, my_object As PHFocusDepnObject)
Returns : Long
Requires :
Modifies :
Effects : Deletes the specified attribute of the focused dependency

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the attribute does not exist
Returns PH_ERROR if an error is encountered

Name : PHDeleteProdActi
Syntax : PHDeleteProdActi (my_object As PHFocusDepnObject, activity_id As Long)
Returns : Long
Requires :
Modifies :
Effects : Deletes the producer activity id from the focused dependency object.

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the relationship does not exist
Returns PH_ERROR if an error is encountered

Name : PHFindAttr
Syntax : PHFindAttr (attr_name As String, attr_value As String, activity_id_list() As Long)
Returns : Long
Requires :
Modifies : activity_id_list
Effects : Finds all activities and bundles which have an attribute attr_name with value attr_value.
          Modifies activity_id_list to be the IDs of all the matching activities and bundles.

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if there are no activities or bundles with the specified attribute/value pair
Returns PH_ERROR if an error is encountered
Name : PHFindDAtr
Syntax : PHFindDAtr (datr_name As String, datr_value As String, depn_id_list() As Long)
Returns : Long
Requires :
Modifies : depn_id_list
Effects : Finds all dependencies which have an attribute datr_name with value datr_value. Modifies depn_id_list to be the IDs of all the matching dependencies.

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if there are no dependencies with the specified attribute/value pair
Returns PH_ERROR if an error is encountered

Name : PHFindUniqueID
Syntax : PHFindUniqueID (entity_type As Integer, unique_id As String)
Returns : Long
Requires :
Modifies :
Effects : Finds the entity id with the specified unique ID. The valid entity_types are as follows:
   PH_ACTIVITY
   PH.Bundle
   PH_DEPENDENCY

Returns the ID of the entity_type which matches the unique ID
Returns PH_DUPLICATE if more than one entity has the same unique ID
Returns PH_NOT_FOUND if no entities match
Returns PH_WRONG_TYPE if an invalid entity_type is specified
Returns PH_ERROR if an error is encountered

Name : PHGetAll
Syntax : PHGetAll (next_type As Integer, my_object As PHFocusActiObject, id_list() As Long)
Returns : Long
Requires :
Modifies : id_list
Effects : Gets all next relationships defined by the type to the focused object. The valid next_types are as follows:
   PH_DECOMPOSITION
   PH.WHERE_USED
   PH_SPECIALIZATION
   PH_GENERALIZATION
   PH_CONSUMER_DEPENDENCY
   PH_PRODUCER_DEPENDENCY
   PH_ANCESTOR_DEPENDENCY
   PH_UNDEFINED_TYPE

Modifies id_list to be the list of ID's of all get next relationships

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if there are no relationships of the specified type
Returns PH_WRONG_TYPE if the specified next_type is invalid
Returns PH_ERROR if an error is encountered
Returns PH_NOT_ACTIVITY if we try to get a decomposition, where_used, producer, consumer, ancestor dependency for a currently focused bundle
Name : PHGetAuncsActi
Syntax : PHGetAuncsActi (my_object As PHFocusDepnObject)
Returns : Long
Requires :
Modifies :
Effects : Returns the activity ID of the least common ancestor of the focused object.

Returns PH_NOT_FOUND if the dependency cannot be found
Returns PH_ERROR if an error is encountered

Name : PHGetContact
Syntax : PHGetContact (entity_type As Integer, entity_id As Long, entity_contact As String
Returns : Long
Requires :
Modifies : entity_contact
Effects : Gets the contact corresponding to the entity_id of type entity_type and modifies the
entity_contact string. If the ID cannot be found, entity_contact is set to "Not found". If an error
is encountered, entity_contact is set to "Error occurred". The valid entity_types are:

PH/activity
PH_BUNDLE
PH_DEPENDENCY

Returns PH_SUCCESS if ID can be found
Returns PH_NOT_FOUND if ID cannot be found
Returns PH_WRONG_TYPE if an invalid entity_type is specified
Returns PH_ERROR if an error is encountered

Name : PHGetFirst
Syntax : PHGetFirst (next_type As Integer, my_object As PHFocusActiObject)
Returns : Long
Requires :
Modifies : my_object
Effects : Iterator to get first relationship defined by the type to the focused object. This resets the iterator
for the PHGetNext function. The valid next_types are as follows:

PH_DECOMPOSITION
PH WHERE USED
PH_SPECIALIZATION
PH GENERALIZATION
PH CONSUMER_DEPENDENCY
PH_PRODUCER_DEPENDENCY
PH ANCESTOR_DEPENDENCY
PH_UNDEFINED_TYPE

Returns the ID of the first activity/bundle/dependency
Returns PH_NOT_FOUND if there are no more, or if the appropriate pointer in my_object is
invalid (has been deleted)
Returns PH_WRONG_TYPE if the specified next_type is invalid
Returns PH_ERROR if an error is encountered
Returns PH_NOT_ACTIVITY if we try to get a decomposition or where_used for a currently
focused bundle
Name : PHGetFirstAttr
Syntax : PHGetFirstAttr (my_object As PHFocusActiObject, attr_name As String, attr_value As String)
Returns : Long
Requires :
Modifies : attr_name, attr_value, my_object
Effects : Gets the first attribute of the focused object and modifies the attr_value and attr_value strings appropriately. It also resets the iterator PHGetNextAttr

Returns PH_SUCCESS if all is O.K.
Returns PH_ERROR if an error is encountered
Returns PH_NOT_FOUND if there are no more attributes or if the attribute pointer is invalid

Name : PHGetFirstConsActi
Syntax : PHGetFirstConsActi (my_object As PHFocusDepnObject)
Returns : Long
Requires :
Modifies : my_object
Effects : Iterator to get first consumer activity id of the focused dependency. This resets the iterator for the PHGetNextConsActi function.

Returns the ID of the first consumer activity
Returns PH_NOT_FOUND if there are no more, or if the appropriate pointer in my_object is invalid (has been deleted)
Returns PH_ERROR if an error is encountered

Name : PHGetFirstDAtr
Syntax : PHGetFirstDAtr (my_object As PHFocusDepnObject, attr_name As String, attr_value As String)
Returns : Long
Requires :
Modifies : attr_name, attr_value and my_object
Effects : Gets the first attribute of the focused dependency and modifies the attr_value and attr_value strings appropriately. It also resets the iterator PHGetNextAttr

Returns PH_SUCCESS if all is O.K.
Returns PH_ERROR if an error is encountered
Returns PH_NOT_FOUND if there are no more attributes or if the attribute pointer is invalid
Name : PHGetFirstEntity
Syntax : PHGetFirstEntity (entity_type As Integer)
Returns : Long
Requires :
Modifies : my_object
Effects : Iterator to get first entity defined by the entity_type in the database. This resets the iterator for
the PHGetNextEntity function. The valid next_types are as follows:
   PH_ACTIVITY
   PH_BUNDLE
   PH_DEPENDENCY

Returns the ID of the first entity_type in the database
Returns PH_NOT_FOUND if there are no more, or if the appropriate pointer in my_object is
invalid (has been deleted)
Returns PH_WRONG_TYPE if the specified entity_type is invalid
Returns PH_ERROR if an error is encountered

Name : PHGetFirstProdActi
Syntax : PHGetFirstProdActi (my_object As PHFocusDepnObject)
Returns : Long
Requires :
Modifies : my_object
Effects : Iterator to get first producer activity id of the focused dependency. This resets the iterator for
the PHGetNextProdActi function.

Returns the ID of the first consumer activity
Returns PH_NOT_FOUND if there are no more, or if the appropriate pointer in my_object is
invalid (has been deleted)
Returns PH_ERROR if an error is encountered

Name : PHGetName
Syntax : PHGetName (entity_type As Integer, entity_id As Long, entity_name As String)
Returns : Long
Requires :
Modifies : entity_name
Effects : Gets the name corresponding to the ID of type entity_type and modifies the entity_name string.
If the ID cannot be found, entity_name is set to "Not found". If an error is encountered,
entity_name is set to "Error occurred". The valid entity_types are:
   PH_ACTIVITY
   PH_BUNDLE
   PH_DEPENDENCY

Returns PH_SUCCESS if ID can be found
Returns PH_NOT_FOUND if ID cannot be found
Returns PH_WRONG_TYPE if an invalid entity_type is specified
Returns PH_ERROR if an error is encountered.
Name : PHGetNext  
Syntax : PHGetNext (next_type As Integer, my_object As PHFocusActiObject)  
Returns : Long  
Requires :  
Modifies : my_object  
Effects : Iterator to get next relationship defined by the type to the focused object. The valid next_types are as follows:  
PH_DECOMPOSITION  
PH_WHERE_USED  
PH_SPECIALIZATION  
PH_GENERALIZATION  
PH_CONSUMER_DEPENDENCY  
PH_PRODUCER_DEPENDENCY  
PH_ANCESTOR_DEPENDENCY  
PH_UNDEFINED_TYPE  

Returns the ID of the next activity/bundle/dependency  
Returns PH_NOT_FOUND if there are no more, or if the appropriate pointer in my_object is invalid (has been deleted)  
Returns PH_WRONG_TYPE if the specified next_type is invalid  
Returns PH_ERROR if an error is encountered  
Returns PH_NOT_ACTIVITY if we try to get a decomposition or where_used for a currently focused bundle

Name : PHGetNextAttr  
Syntax : PHGetNextAttr (my_object As PHFocusActiObject, attr_name As String, attr_value As String)  
Returns : Long  
Requires :  
Modifies : attr_name and attr_value  
Effects : Iterator that gets the next attribute of the focused object and modifies the attr_value and attr_value strings appropriately.  

Returns PH_SUCCESS if all is O.K.  
Returns PH_ERROR if an error is encountered  
Returns PH_NOT_FOUND if there are no more attributes or if the attribute pointer is invalid

Name : PHGetNextConsActi  
Syntax : PHGetNextConsActi (my_object As PHFocusDepnObject)  
Returns : Long  
Requires :  
Modifies : my_object  
Effects : Iterator to get the next consumer activity of the focused dependency  

Returns the ID of the next consumer activity  
Returns PH_NOT_FOUND if there are no more, or if the appropriate pointer in my_object is invalid (has been deleted)  
Returns PH_ERROR if an error is encountered

72
Name  :  PHGetNextDAtr
Syntax :  PHGetNextDAtr (my_object As PHFocusDepnObject, attr_name As String, attr_value As String)
Returns :  Long
Requires :
Modifies :  attr_name and attr_value
Effects :  Iterator that gets the next attribute of the focused object and modifies the attr_value and attr_value strings appropriately

Returns PH_SUCCESS if all is O.K.
Returns PH_ERROR if an error is encountered
Returns PH_NOT_FOUND if there are no more attributes or if the attribute pointer is invalid

Name  :  PHGetNextEntity
Syntax :  PHGetNextEntity (entity_type As Integer)
Returns :  Long
Requires :
Modifies :  my_object
Effects :  Iterator to get the next entity as defined by the entity_type. The valid entity_types are as follows:
    PH_ACTIVITY
    PH_BUNDLE
    PH_DEPENDENCY

Returns the ID of the next entity_type
Returns PH_NOT_FOUND if there are no more, or if the appropriate pointer in my_object is invalid (has been deleted)
Returns PH_WRONG_TYPE if the specified entity_type is invalid
Returns PH_ERROR if an error is encountered

Name  :  PHGetNextProdActi
Syntax :  PHGetNextProdActi (my_object As PHFocusDepnObject)
Returns :  Long
Requires :
Modifies :  my_object
Effects :  Iterator to get the next producer activity of the focused dependency

Returns the ID of the next consumer activity
Returns PH_NOT_FOUND if there are no more, or if the appropriate pointer in my_object is invalid (has been deleted)
Returns PH_ERROR if an error is encountered
Name : PHGetUniqueID
Syntax : PHGetUniqueID (entity_type As Integer, entity_id As Long, entity_unique_id As String)
Returns : Long
Requires :
Modifies : entity_unique_id
Effects : Finds the unique_id corresponding to the ID of type entity_type and modifies the
entity_unique_id string. If the ID cannot be found, entity_unique_id is set to "Not found". If an
error is encountered, entity_unique_id is set to "Error occurred". The valid entity_types are:
   PH_ACTIVITY
   PH_BUNDLE
   PH_DEPENDENCY

   Returns PH_SUCCESS if ID can be found
Returns PH_NOT_FOUND if ID cannot be found
Returns PH_WRONG_TYPE if an invalid entity_type is specified
Returns PH_ERROR if an error is encountered.

Name : PHMoveSpec
Syntax : PHMoveSpec (child_id As Long, old_parent As Long, new_parent As Long)
Returns :
Requires :
Modifies :
Effects : Moves a specialization in the hierarchy. child_id is moved from the specialization of old_parent
to new_parent. Attributes that were inherited from the old parent but are not present in the new
parent are retained by the move operation.

   Returns PH_SUCCESS
Returns PH_NOT_FOUND if the specified old specialization does not exist
Returns PH_NOT_FOUND if the specified child_id does not exist
Returns PH_DUPLICATE if the specified new specialization already exists
Returns PH_ERROR if an error is encountered

Name : PHNewActi
Syntax : PHNewActi (activity_name As String, contact As String, unique_id As String)
Returns : Long
Requires :
Modifies :
Effects : Adds a new activity to the database with the specified activity_name and contact. Also sets a
unique ID for the activity. If the unique ID is the empty string, then it creates a unique ID.

   Returns the new activity ID if all is O.K.
Returns PH_ERROR if an error is encountered.
**Name**: PHNewBund  
**Syntax**: PHNewBund (activity_name As String, contact As String, unique_id As String)  
**Returns**: Long  
**Requires**:  
**Modifies**:  
**Effects**: Adds a new bundle to the database with the specified bundle_name and contact. Also sets a unique ID for the activity. If the unique ID is the empty string, then it creates a unique ID.  
Returns the new bundle ID if all is O.K.  
Returns PH_ERROR if an error is encountered.

**Name**: PHNewDepn  
**Syntax**: PHNewDepn (dependency_name As String, contact As String, unique_id As String, dependency_type As Integer)  
**Returns**: Long  
**Requires**:  
**Modifies**:  
**Effects**: Adds a new dependency of the specified type to the database with the specified dependency_name and contact. Also sets a unique ID for the activity. If the unique ID is the empty string, then it creates a unique ID. The valid types are:  
- PH_DEPN_P_C = Producer/Consumer  
- PH_DEPN_PS_C = Producers/Consumer  
- PH_DEPN_P_CS = Producer/Consumers  
- PH_DEPN_PS_CS = Producers/Consumers  
- PH_DEPN_SHARED_RESOURCE = Shared Resource  
- PH_DEPN_COMMON_OBJECT = Common Object  
By default, sets the most least ancestor of this dependency to be the root i.e., 0  
Returns the new dependency ID if all is O.K.  
Returns PH_ERROR if an error is encountered.  

NOTE: To propagate the inheritance of dependencies, it is necessary to first set all producer and consumer activities, set the least common ancestor and then explicitly call the function Dependency_Added from the inheritance layer.

**Name**: PHOpenDB  
**Syntax**: PHOpenDB (db_name As String)  
**Returns**: Long  
**Requires**: db_name is a valid filename for a Process Handbook database  
**Modifies**:  
**Effects**: Opens the database and checks to see that there is a root with ID 0. If not it adds it (used for boot-strapping an empty database).  
Returns PH_SUCCESS if all is O.K.  
Returns PH_ERROR if the database cannot be found or if an error is encountered.
**Name**: PHSetAttr

**Syntax**: PHSetAttr (my_object As PHFocusActiObject, attr_name As String, attr_value As String, inheritance_switch As Integer, context_ids() As Long, new_acti_ids() As Long)

**Returns**: Long

**Requires**: When the attribute is to be set in the context of a specific decomposition, context_ids is a bottom up array of activity ids representing the context in which the set is to be made.

context_ids and new_acti_ids are different arrays, unless there is no context, in which case both context_ids and new_acti_ids should be set to the global constant PH_NO_CONTEXT()

**Modifies**: new_acti_ids

**Effects**: Updates or creates a new attribute slot attr_name with value attr_value. If inheritance switch is TRUE, then the creation of a new slot will cause the slot to be inherited by all its specializations.

new_acti_ids is modified to be the bottom up list of newly specialized activity ids created by the context mechanism.

Returns PH_SUCCESS if all is O.K.
Returns PH_ERROR if an error is encountered

---

**Name**: PHSetDAtr

**Syntax**: PHSetDAtr (attr_name As String, attr_value As String, my_object As PHFocusDepnObject)

**Returns**: Long

**Requires**: 

**Modifies**: 

**Effects**: Updates or creates a new attribute slot attr_name with value attr_value

Returns PH_SUCCESS if all is O.K.
Returns PH_ERROR if an error is encountered

---

**Name**: PHSetFocusActi

**Syntax**: PHSetFocusActi (my_object As PHFocusActiObject, activity_id As Long)

**Returns**: Long

**Requires**: 

**Modifies**: my_object

**Effects**: Initializes the required pointers in my_object. This function must be called prior to many other function calls for activities and bundles

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the specified activity_id cannot be found
Returns PH_ERROR if an error is encountered
Name: PHSetFocusDepn
Syntax: PHSetFocusDepn (my_object As PHFocusDepnObject, dependency_id As Long)
Returns: Long
Requires:
Modifies: my_object
Effects: Initializes the required pointers in my_object. This function must be called prior to many other function calls for dependencies.

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the specified dependency_id cannot be found
Returns PH_ERROR if an error is encountered

Name: PHValueAttr
Syntax: PHValueAttr (my_object As PHFocusActiObject, attr_name As String, attr_value As String)
Returns: Long
Requires:
Modifies: attr_value
Effects: Modifies the attr_value string to be the value of the specified attribute. If the attribute cannot be found, it is set to "Not found". If there is an error, the string is set to "Error occurred".

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the attribute cannot be found
Returns PH_ERROR if an error is encountered

Name: PHValueDAtr
Syntax: PHValueDAtr (my_object As PHFocusDepnObject, attr_name As String, attr_value As String)
Returns: Long
Requires:
Modifies: attr_value
Effects: Modifies the attr_value string to be the value of the specified attribute. If the attribute cannot be found, it is set to "Not found". If there is an error, the string is set to "Error occurred"

Returns PH_SUCCESS if all is O.K.
Returns PH_NOT_FOUND if the attribute cannot be found
Returns PH_ERROR if an error is encountered