An Integrated Dependency Editor for the Process Handbook

by

Zia Ahmed

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

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 1998

© Zia Ahmed, 1998.
All rights reserved.

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

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

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

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

Eng.

An Integrated Dependency Editor for the Process Handbook

by

Zia Ahmed

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

Abstract

The Process Handbook project at the MIT Center for Coordination Science aims to demonstrate the feasibility of an on-line handbook of business processes. From the end-user perspective, the Process Handbook supports a variety of viewers, each of which emphasizes a different view of a business process. This thesis discusses the design, implementation and functionality of the Dependency Editor, which relates to the dependencies which exist between various organizational processes and the different coordination activities which manage these dependencies. The system provides a user friendly interface which enables users to easily lay out a process representation on a two-dimensional plane, using for building blocks either (1) existing processes within the system, or (2) user-defined processes, which can be created by the system. Two powerful representation metaphors are available: (1) processes can be essentially "opened" up to lay bare the processes and inter-process dependencies which exist in their decomposition, and (2) dependencies can be replaced with their managing processes, revealing the particular surface structure of the process being studied. These two features, together with simple yet powerful drag-drop editorial capabilities provided in conjunction with the rest of the system, make the Dependency Editor a useful tool for viewing or editing processes at various levels of decomposition. All in all, the Dependency Editor contributes towards making the Process Handbook a useful, extensible, and robust tool for analyzing processes.

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

Acknowledgments

I would like to thank many people for their invaluable support and help throughout my college life. Firstly, I would like to thank Professor Thomas Malone, who has been the best thesis supervisor and mentor one could hope for. My immeasurable thanks to John Quimby, trusted counselor and guide, who has nurtured and led me through my graduate year. Thanks to Avi Bernstein, without whose constant help on matters technical or otherwise this project would never have been possible. Thanks also to all the other CCS members who made it such a pleasant place to work – George Herman, Mark Klein and Henry Tang.

On a personal note, I would like to thank all of my friends here at MIT who have made my college years so fulfilling. A special thank you to Abby for her excellent proof-reading and moral support. Thanks also to Umar for introducing me to CCS and for his steadfast friendship these past few years. Last but most importantly, my heartfelt thanks to my parents who have encouraged me at every step, and helped me become the person that I am.

Contents

1 Introduction **8**

2 The Process Handbook **11**

2.1 Introduction to Coordination Theory 11

2.2 System Overview 12

2.2.1 Specialization 13

2.2.2 Decomposition 14

2.2.3 Dependencies, Ports and Connectors 15

2.2.4 Bundles and Navigational Nodes 18

2.2.5 Attributes 19

2.2.6 Inheritance 20

3 The Object Server **22**

3.1 Three-Tier Client/Server Architecture 22

3.2 Types of Objects 24

3.2.1 Entity 24

3.2.2 Relation 25

3.2.3 Attribute 26

3.2.4 PHDB 26

3.3 Extensions to the Object Server 26

3.3.1 Caching 26

3.3.2 Addition of Slots 28

3.3.3 Modification of Connectors 30

3.3.4 Support for Managing Processes 32

4 The Dependency Editor **36**

4.1 Introduction and Motivation 36

4.2 Visual Metaphors 37

4.3 Base Implementation 40

4.3.1 Object Types and Methods 41

4.3.2 Screen Layout 44

4.3.3 Interaction of Objects 44

4.4 User Interface Level Functionality 46

4.4.1 Getting the Decomposition of an Entity 46

4.4.2 Creating Connectors 46

4.4.3 Creating Dependencies between Activities 47

4.4.4 Drag-Drop Operations 49

4.4.5 User Toolbox 49

4.4.6 Setting the Managing Resource for Dependency 49

4.4.7 Setting a Managing Activity for a Dependency 50

4.4.8 Replacing a Dependency with its Managing Activity 50

4.4.9 Replacing an Activity with its Specialization 51

4.5 Integration with the System 52

5 Using the Dependency Editor **54**

5.1 A Process Modeling Example 54

5.1.1 The *Replenish Inventory* Process 54

5.1.2 The *Replenish Inventory* Process in the Process Handbook . . 55

5.2 Enhancing an Existing Process Description 59

6 Future Work **63**

6.1 Theoretical Constructs 63

6.2 Advanced Functions 65

7 Conclusion **66**

List of Figures

2-1 One Level of the Specialization Hierarchy 13

2-2 Decomposition of the *Make Coffee* Process 14

2-3 The Navigational Compass 15

2-4 Two Concurrent Views of Specialization and Decomposition 15

2-5 Three basic types of dependencies among activities 16

2-6 A simple flow dependency 16

2-7 Horizontal and Vertical Connectors 17

2-8 Bundles Under *Sell something* 18

2-9 Tradeoff Matrix for *Sell how?* 19

2-10 Decomposition Inheritance for Sales Processes 20

3-1 The Three Tiers of the Process Handbook 23

3-2 Getting the Specializations of an Entity 27

3-3 Types of Specialization Relations 28

3-4 Decomposition Slots 29

3-5 Slot Inheritance 29

3-6 Slot Path Uniqueness 30

3-7 Slot Aware Connectors 31

3-8 Port Matching for a Managed Flow Dependency 32

3-9 Setting the Managing Process for a Dependency 35

4-1 Visual Representation of Various Types 37

4-2 *Click-and-Replace* Representation of *Make Coffee* 38

4-3 *Worlds-within-Worlds* Representation of *Make Coffee* 39

4-4 Creating a Dependency at Arbitrary Levels in a Decomposition . . . 47
4-5 Dependency at an Arbitrary Level in a Decomposition 48
4-6 Setting the Managing Activity for a Dependency 50
4-7 Replacing a Dependency with its Managing Activity 51

5-1 *Replenish Inventory (Retailer)* Diagram 55
5-2 *Take Delivery (Retailer)* Diagram 56
5-3 Sub-processes in the Decomposition of *Replenish Inventory* 56
5-4 Single Producer-Consumer Connections in *Replenish Inventory* 57
5-5 A Complete View of *Replenish Inventory* 58
5-6 Complete Specifications of the *Replenish Inventory* Process 59
5-7 The *Take Delivery* Process 59
5-8 Enhancing the *Take Delivery* Process 60
5-9 Adding a Dependency 61
5-10 Simple Dependencies 61
5-11 Replacing a Dependency with its Managing Activity 61
5-12 Complete View of the *Replenish Inventory* Process 62

6-1 A Flow Dependency 63
6-2 Replacing a Dependency with its Managing Activity 64
6-3 Replacing a Dependency: Alternate View 64

Chapter 1

Introduction

The Process Handbook project at the MIT Center for Coordination Science involves collecting examples of how different organizations perform similar processes, and organizing these processes in an on-line Process Handbook[9]. The Process Handbook is intended to help people: (1) redesign existing organizational processes, (2) invent new organizational processes, (3) learn about organizations, and (4) automatically generate software to support organizational processes. The methodology used in the Process Handbook is semantically very rich. It allows users to represent and analyze complex processes, which can also be viewed in many other common process representation formats, for example, IDEF0, data flow, etc.

The Process Handbook methodology uses concepts from coordination science and computer science such as those of objects and inheritance. All processes in the Process Handbook are categorized in a specialization hierarchy, with very generic processes at the top and increasingly specialized processes at the lower levels. Users of the Handbook can use the specialization hierarchy to understand the deep structure of processes and to invent more specialized versions of existing processes. The more complex processes in the Handbook have other simpler processes in their decomposition. This allows users to view any process at an arbitrary level of detail. A process in the Handbook can also have any number of attributes which define the key characteristics of the process. The specializations of a process inherit both its attributes and its decomposition. Finally, the Process Handbook methodology uses the notion

of *coordination* as the act of managing dependencies between different processes.

In implementation terms, the Process Handbook has been through a number of cycles of varying codebases developed by many software engineers. The current version of the Process Handbook is highly modular and abstract, and is based on a three-tier client-server architecture. The bottom tier is that of the physical database, which stores process representations in a logical schema. The middle tier is a transparent API in the form of an Object Server, which exposes OLE (Object Linking and Embedding) objects to the clients and provides various methods for their manipulation and mutation. The API abstracts away the details of complex algorithms involved in process manipulation and also the details of the database schema. The third tier is concerned with Graphical User Interface (GUI) clients for end users of the Handbook. Users interact with the Handbook through these clients, which is why clarity, simplicity and elegance are essential goals GUI design. This thesis – though primarily concerned with a GUI client for the Handbook – touches on all three tiers (i.e. database, object API, and user interface).

Many scientists at the Center for Coordination Science – especially John Quimby and Abraham Bernstein – took an active part in the design of this system. References to 'we' throughout this document should be taken to mean the above mentioned people along with the author. A substantial prototype of the Dependency Editor was implemented by Avi Bernstein in the summer of 1997.

This thesis is organized in a readily readable format. Chapter 2 narrates some detail about the actual semantics of the Process Handbook, and gives a brief discussion of coordination theory. Chapter 3 relates to the Process Handbook Object Server, and reveals some necessary extensions which were made to the server in order to support the functionality of the Dependency Editor. Chapter 4 discusses the motivation for the Dependency Editor, and gives a high-level view of its design, implementation and functionality. Chapter 5 is written in a user-manual format, which users already familiar with the Process Handbook can reference to utilize the extended functions provided by the Dependency Editor. This chapter illustrates a real-world process modeling example. Future work – including possible extensions to the Dependency

Editor and the Process Handbook – is discussed in Chapter 6. Finally, the conclusions drawn from this project are presented in Chapter 7.

Chapter 2

The Process Handbook

2.1 Introduction to Coordination Theory

Some grasp of coordination theory is necessary to comprehend the methodology and semantics of the Process Handbook. Coordination science is the underlying philosophy of the Process Handbook. Much of the terminology used in the context of the Handbook is borrowed from coordination-theoretic concepts.

Coordination theory is an interdisciplinary study of coordination. Coordination is broadly defined as the action of working together. In specific coordination theoretic terms, coordination is the act of managing dependencies between various processes. Research within this area draws heavily from such varied fields as computer science, organization theory, operations research, economics, linguistics, and psychology[8].

A coordination theoretic analysis of a high-level process requires four major elements: (1) *goals* are the desired results of the process; in order to achieve these goals, it may be necessary for a collection of (2) *actors* to perform a set of (3) *activities*, in some temporal or spatial order. An implicit assumption made in the coordination perspective is that these activities are related in some sense; that is, there exists a well-defined set of (4) *dependencies* between them. Coordination theory is based on the art of managing the dependencies between these activities with other activities, with the intent of achieving the given set of goals.

The Process Handbook uses the more formal view of coordination, that is: man-

aging dependencies between activities. Processes in the Process Handbook can be decomposed into sub-processes and the dependencies between these sub-processes. The dependencies between these sub-processes can be managed by managing activities – processes that manage dependencies. The act of managing a dependency with a managing process may in turn introduce additional dependencies. Being able to view and alter the managing process for a dependency gives users of the Process Handbook the power to invent new processes, understand existing processes in more detail, and improve current processes. In the context of dependency editing, dependencies and their managing activities are considered to be semantically as important as sub-processes in a decomposition.

The Process Handbook can be used to invent new processes in two fundamentally different ways. Using the *Top-Down* approach, (1) a user must specify the high level goals (parent processes), (2) these processes are decomposed into sub-processes, (3) the dependencies between the various sub-processes are specified, (4) the managing activities for these dependencies are specified. The last step specifies how *coordination* between the sub-processes takes place. This process is performed recursively to yield finer detail of the parent process. The *Bottom-Up* approach reverses the order of the first two steps. Essentially, the user first specifies the finest resolution of the process and iteratively uses the relatively simple sub-processes to build up an increasingly complex process description.

2.2 System Overview

The objective of the Process Handbook project is to prove the feasibility and utility of populating an on-line handbook of business processes (see [9] for more detail). A great deal of research is involved in collecting and organizing the data which finally becomes part of the Handbook's growing database. Field data is collected to show how different organizations perform the same processes in the context of their high-level goals. An interesting side-effect of this method is that tradeoffs between slightly different processes immediately become apparent. The Handbook is essentially a tool

for managers, workers or consultants who are interested in any level of organization redesign. The Handbook is also an invaluable aid to pure theoreticians, in the sense that it can help them to simulate abstract organizational structures and models which have not yet been realized in practice.

2.2.1 Specialization

The Process Handbook methodology borrows the concepts of specialization and inheritance from object-oriented languages in the realm of computer science. Essentially, this means that a process can be specialized into a finer refinement, which may differ from its parent process by some measure. Thus, all processes in the Handbook can be organized in a so-called *specialization hierarchy.* All activities are nodes in the specialization hierarchy, rooted at the most general activity namely *Act.* The Handbook

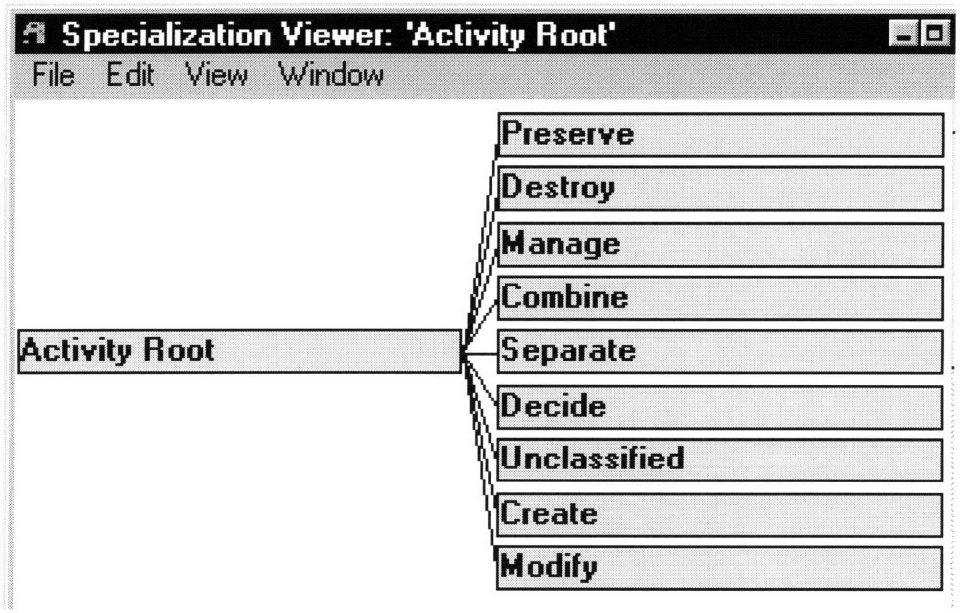

Figure 2-1: One Level of the Specialization Hierarchy

also involves the notion of inheritance; processes lower down in the specialization hierarchy inherit the attributes of their parent processes, much like how a sub-class in an object-oriented language inherits the properties of its superclasses. This lends enormous power to the Handbook in terms of ease of process engineering; instead

of reinventing the wheel each time a user needs to design or re-engineer a process, she may just create a specialization of an existing process and perform the necessary changes. Thus, once a suitably representative specialization hierarchy has been defined, process creation – in a large part – becomes a matter of successive refinements to an already existing entry in the Handbook. An activity in the Process Handbook can have many specializations and generalizations. A general *make coffee* process can have the activity *make ice coffee* as a specialization and the activity *make beverage* as a generalization.

2.2.2 Decomposition

In addition to the specialization view, the Handbook offers the view of decomposition of a process into recursively simpler processes. Processes can thus be decomposed into sub-processes. For example, the *make coffee* process referred to in 2.2.1 may in fact be composed of the simpler processes *gather ingredients*, *brew coffee* and *serve coffee*. This decomposition allows users to view a high-level process in varying levels of detail. In theory, a process should be infinitely decomposable, but since the Handbook can only have finite content, it is up to the user to specify the amount of detail desired in a process description.

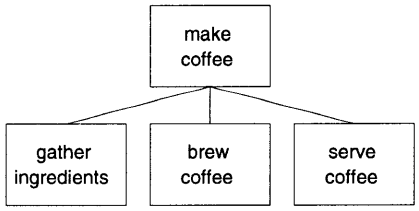

Figure 2-2: Decomposition of the *Make Coffee* Process

The two dimensions of decomposition and specialization come together to form the so-called *Navigation Compass*. The north-south dimension of the compass refers to composition- decomposition respectively, while the east-west dimension of the compass refers to specialization-generalization. In this sense, a process can be successively

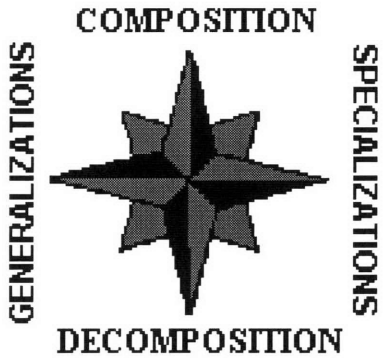

Figure 2-3: The Navigational Compass

refined when a sub-process in its decomposition is replaced by a specialization which is more relevant to that process than its parent. For example, in Figure 2-4, the *make ice coffee* process is a specialization of the *make coffee* process. However, the *serve coffee* process in the decomposition of *make coffee* has been refined to the *serve coffee with ice* process, hence making *make ice coffee* a pertinent refinement of the original *make coffee* process.

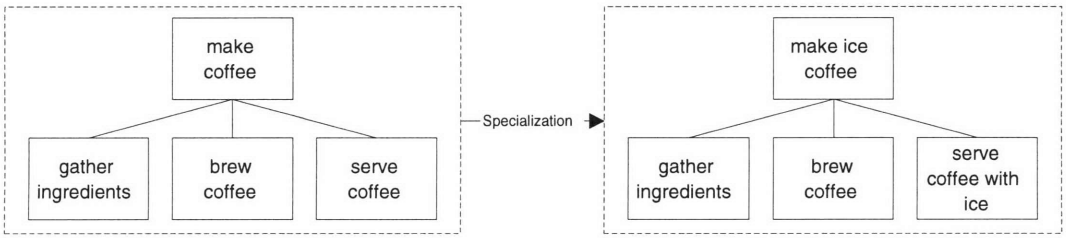

Figure 2-4: Two Concurrent Views of Specialization and Decomposition

These two views together lend great expressive power to the Process Handbook semantics.

2.2.3 Dependencies, Ports and Connectors

An essential element of the Process Handbook methodology are dependencies. Dependencies represent the dependence of activities upon one another. Usually, depen-

dencies enforce some form of temporal or resource constraint upon their connecting processes. There are three basic kinds of dependencies: (1) *flow* dependencies arise when a resource produced by one activity is consumed by another activity, (2) *sharing* dependencies arise when a single resource is consumed by multiple activities, and (3) *fit* dependencies occur when multiple activities produce a single resource[9].

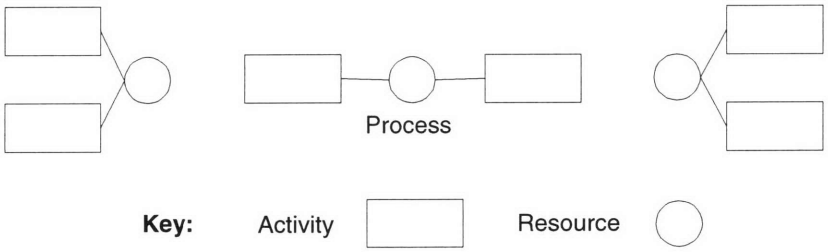

Figure 2-5: Three basic types of dependencies among activities

A simple flow dependency is illustrated in Figure 2-6. In the decomposition of the *make coffee* process, there exists a dependency between the *gather ingredients* and *brew coffee* sub-processes. Even this simple dependency represents quite a few constraints on the processes: (1) a *prerequisite* or *temporal* constraint; ingredients must be gathered before the coffee can be brewed, (2) *accessibility* constraint; the beans must be transported to the place where the coffee will be brewed, and (3) *usability* or *compatibility* constraint; the brewing mechanism should be able to use the coffee beans that are fetched. Dependencies which interact with many producers and consumers can have much more complexity in their semantic information.

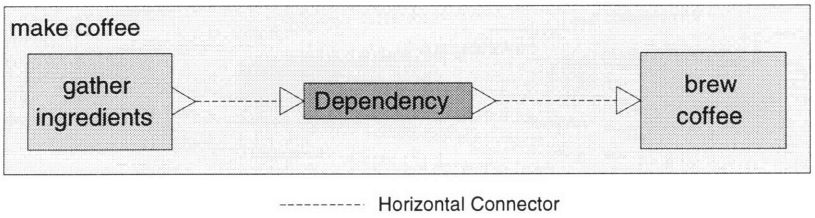

Figure 2-6: A simple flow dependency

Activities and dependencies in the Handbook semantics have input and output ports. As illustrated in Figure 2-6, the triangles on the activities and the dependency are their ports. In this particular example, *gather ingredients* has an output (producer) port that is an outlet for the gathered ingredients. Similarly, *brew coffee* has an input (consumer) port through which the ingredients are consumed. Finally, the flow dependency has ports that are used in the mediation of the coffee beans that flow from *gather ingredient* to *brew coffee*. Ports of entities lie in their decomposition. A high-level parent entity is related to sub-entities and ports in its decomposition via decomposition relations.

Ports of different activities and dependencies are connected to each other by connectors. Connectors represent simple links between ports, indicating flows. Connectors in the Handbook semantics are of two sorts: (1) *horizontal connectors* connect the ports of immediate siblings in a decomposition, and (2) *vertical connectors* connect the ports of an activity and the ports of the sub-activities in its decomposition. Vertical connectors are instrumental in defining dependencies between arbitrarily deep sub-processes. Horizontal connectors are shown in figure 2-6 as lines between ports. Vertical connectors are shown in figure 2-7, which is essentially a refinement of Figure 2-6.

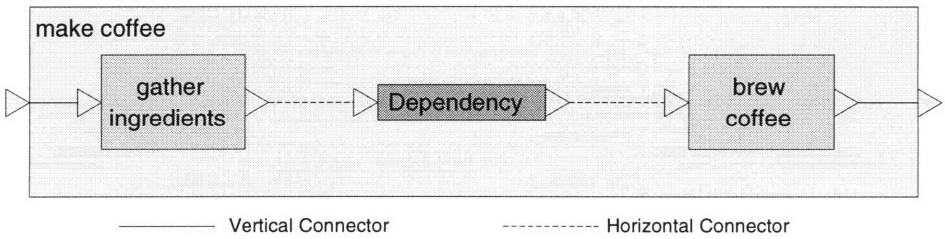

Figure 2-7: Horizontal and Vertical Connectors

The Handbook defines coordination as the act of managing dependencies between activities. Thus, an important part of a dependency is its managing activity. Essentially, a dependency is a high-level abstraction for the managing activity and other possibly simpler dependencies. Therefore, another important aspect of the Handbook

functionality is that users are able to set the managing activity for a dependency from a pre-defined class of managing activities or an activity of the user's own creation. The original dependency is broken into more dependencies as a result of setting a managing activity.

2.2.4 Bundles and Navigational Nodes

The specialization hierarchy possesses a powerful grouping construct called a *bundle*. All specializations of a given process are arranged in bundles of specialized processes.

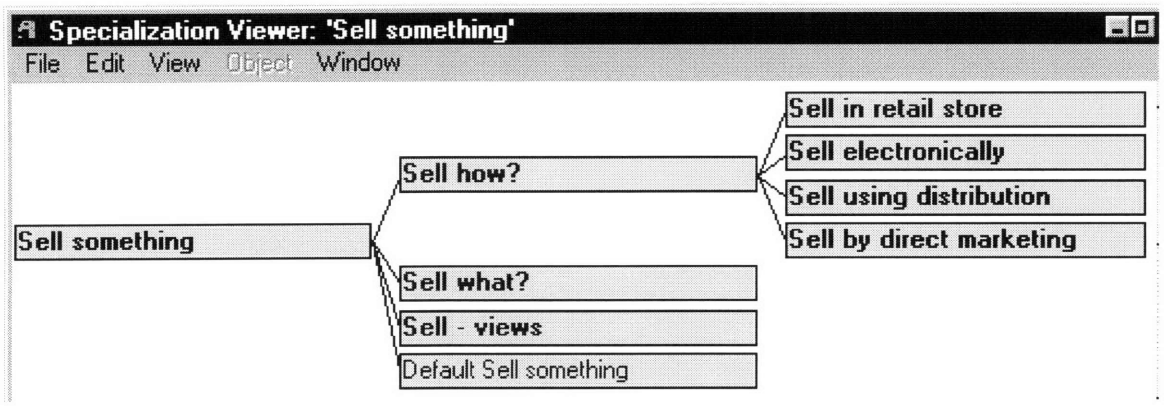

Figure 2-8: Bundles Under *Sell something*

Figure 2-8 illustrates all the bundles found under the activity *Sell something*. The specialization tree is expanded to reveal the activities under the bundle *Sell how?*. Bundles are used to compare alternatives between different processes. These alternatives are usually presented as a tradeoff matrix generated from the attributes of all the processes under a single bundle. Bundles are also used for restricting certain kinds of inheritance.

Figure 2-9 illustrates the tradeoff matrix for the bundle *Sell how?*. The matrix presents the attributes of all the activities under the bundle in an easily perusable format. A user of the Handbook can now look at the matrix and choose the process which best suits her needs depending on which of the attributes of the process under discussion are most useful for her purpose.

Activity Details: Sell how?

File Subactivities Used By Generalizations Specializations

	Cost of selling	Time to sell	Quality of service	Suggested products
Sell electronically	Low	Fast	Medium	Commodities
Sell using distribution channels	Low	Medium	Medium	Commodities
Sell by direct marketing	Low	Medium	Low	Specialty items
Sell in retail store	Medium	Medium	Medium	Low margin commodities

Figure 2-9: Tradeoff Matrix for *Sell how?*

Navigational nodes can be thought of as folders for storing references to similar activities. Navigational nodes are related to other Handbook entities through navigational relations. The concept of navigational nodes is very similar to that of folders of bookmarks in a Web browser. Each folder – or navigational node – of bookmarks can contain bookmarks (references to entities) and other folders (other navigational nodes). Navigational nodes can be decomposed and specialized. The specializations of a navigational node inherit its decomposition, attributes, and navigational relations. A navigational node can only have other navigational nodes in its decomposition.

2.2.5 Attributes

Continuing with the object-oriented analogy, all entities in the Process Handbook have attributes. These attributes are inherited in an object-oriented fashion, as discussed in 2.2.6. The attributes of an entity distinguish it from other entities; for example *Name*, *ID*, *PIFID* (Process Interchange Format[5] ID) etc. are distinguishing characteristics of every entity in the Process Handbook. Attributes may be read-only or read-write; for example, the *ID* attribute is unique for an entity and cannot be modified by a user once an entity has been created. Some attributes are generated by the system itself, and these attributes may or may not be read-only. Other attributes help in explaining their reference entity; for example, the attribute *Actors involved*

in an activity stores the actors which are required for a certain activity to occur.

2.2.6 Inheritance

The Process Handbook is an object-oriented system in the sense that it provides full functionality for inheritance of entities. This functionality of the Handbook considerably increases its representational power and value to users. All specializations of an entity inherit its attributes and decomposition. Whereas traditional object-oriented systems have a hierarchy of objects where the specialized objects inherit the functionality of the more general objects, the Process Handbook has a hierarchy of both verbs (activities) and objects (the resources in the Process Handbook specialization hierarchy are actually objects) that behave in a similar fashion.

Entity specializations inherit the attributes of the parent entity in a variety of ways, depending on the attribute type. An attribute may not be inherited, or it may be inherited with a default value. An attribute may also be inherited without a value at all. In the normal case however, an attribute is inherited along with its value.

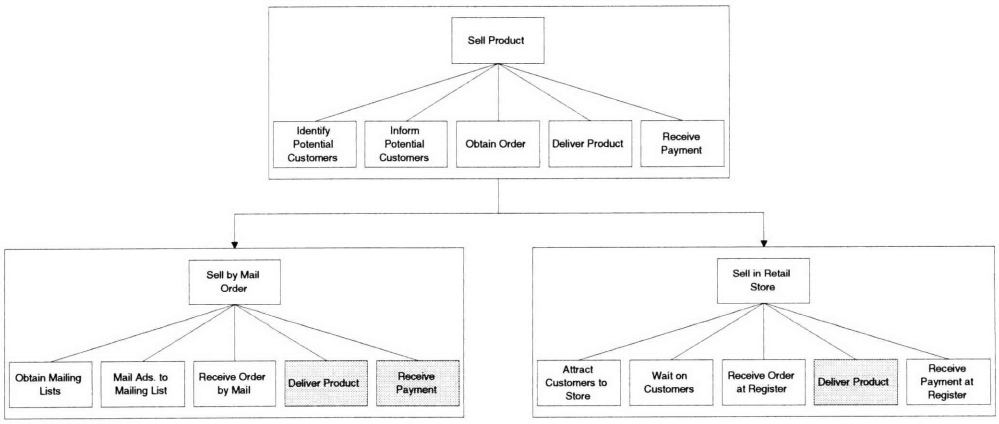

Figure 2-10: Decomposition Inheritance for Sales Processes

Entity specializations also inherit the decomposition of their parent entity. This process involves the inheritance of sub-activities, dependencies, ports and connectors which may exist in the decomposition of the parent entity. Of course, the sub-

activities in the specialized processes may be refined or replaced with completely different processes to give meaningful differences from the parent process. Figure 2-10 illustrates this concept; *Sell by Mail Order* and *Sell in Retail Store* are the specializations of the generic process *Sell Product.* The shaded sub-activities are inherited without change[9].

Default inheritance mechanisms are by reference only; that is, if an entity A has another entity B in its decomposition, then a newly created specialization C of A will also have the same B in its decomposition. The question arises, what happens if a user makes a change to B while looking at C's decomposition? Since the inheritance is by reference, a first answer would be that the entity B is changed absolutely and this change is also reflected in the decomposition of entity A. This approach is fundamentally flawed: consider the scenario where processes A and C are owned by different users, and process B is a generic entity which is also being used concurrently in the decomposition of other entities throughout the Handbook. Now, users are concerned with their own processes and would not want their process to change just because another user finds it useful to modify her process description.

This argument leads to the idea of a context in a decomposition view. Changes to entities in the Handbook are hence always in the context of some decomposition. In the scenario above, the moment a user decides to edit B in the context of C, the system generates a new specialization B' of B, which is identical to B up to the point of the edit. The decomposition pointer in A still points to the old, unchanged B while C now has B' in its decomposition.

Chapter 3

The Object Server

3.1 Three-Tier Client/Server Architecture

The current version of the Process Handbook is implemented as a three-tier architecture. This makes the implementation exceedingly modular and abstracts away much of the complexity of lower tiers. Also, the client-server architecture enforces hard modularity, which ensures that errors in one tier do not interact with other tiers or cause them to crash. This is a very useful architecture – consider the case where one tier of the architecture crashes; the tier above will receive a helpful error, which it can propagate upwards to yield a final error message to the user. The highest level client will not crash merely because one of the lower levels crashes. Figure 3-1 illustrates the three tiers of the Handbook.

The bottom tier is the physical database, which stores the Handbook data in a logical schema. Notice that the database implementation itself may be tiered. Currently, the Handbook uses an MS-Access database. In the foreseeable future, it may be justifiable to switch to a better database technology such as Oracle or Informix. However, the current size of the Handbook database does not require such sophisticated technology. Abstraction ensures that the higher level tiers are not concerned with the implementation of the physical database. So, replacing the database technology should be a fairly simple task.

The middle tier is the Process Handbook Object Server[4] which exposes OLE

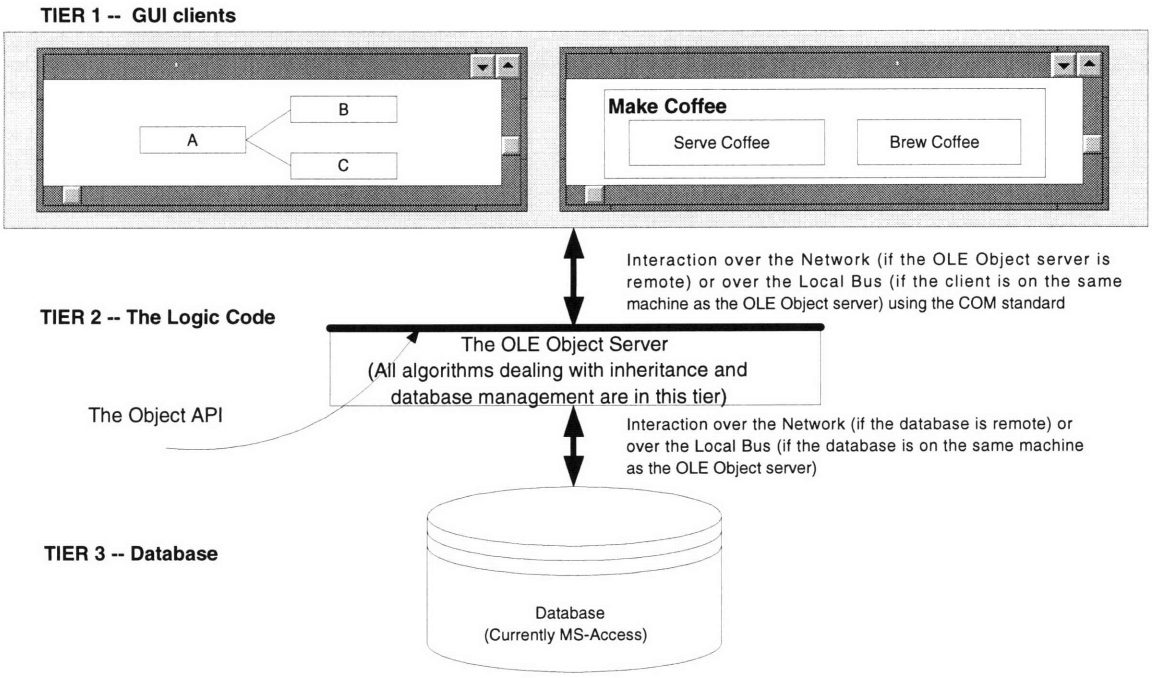

Figure 3-1: The Three Tiers of the Process Handbook

objects to the clients through a thin and transparent API. This tier handles all structural and inheritance algorithms. Once again, hard modularity in the design ensures that no user error is propagated to the database layer. Hence, the database cannot be unwittingly corrupted by an inexperienced user. The API also abstracts away algorithmic concerns from the clients. For example, if the client were to add a specialization to an entity, it would just ask the Object Server to do so – the API would then take care of inheriting all the attributes and the decomposition of the entity to the newly created specialization. Abstraction once again reduces the system definition to an interface specification. The object server can be re-implemented without any change to the clients. In this sense, the API specifications suffice to give its correct definition.

The top-level tier of this architecture consists of GUI clients for the Process Handbook. From the end-user perspective, this tier is the most important in some sense, since it is through the GUI that a user will be able to interact with the system. GUIs are built on top of the object API and are hence simple in the sense that they do

not have to deal with structural and inheritance algorithms, which are handled by the middle-tier. The Dependency Editor is one such GUI which offers a dependency and coordination-intensive perspective to the end-user about existing processes in the Handbook.

One useful property of the three-tier architecture is that it is defined by the DCOM[1] standard. Thus, it is entirely feasible for the clients, server and database to be resident on three different machines on a network and communicating via standard network communication protocols such as TCP/IP. The most feasible implementation which is realizable in practice is that of the server and database to reside on the same machine, since this is where most of the network traffic is to be expected. Clients can be on different machines scattered across a network, and would display processes in their native representation by querying the server for data and updates.

3.2 Types of Objects

A brief discussion of the object server and the types of objects it exposes to the clients is now presented. Details of actual implementation algorithms will be provided in a high-level perspective.

3.2.1 Entity

The *Entity* object is the most general type of object exposed by the server. An entity represents any of the existing items in the Handbook. The *type* property of an entity object further specifies what type the entity actually is. Entities can be of the following types: *activity*, *dependency*, *resource*, *port*, *navigational node* or *bundle*. Each of these specific types of entities have their own properties and attributes. Entities have *name*, *id*, *contact*, and *unique_id* properties. Entity ids are unique in the Handbook and provide a handle to the abstract entity object.

The entity object provides several manipulation methods to the client. The most commonly used methods are *AddNewSpecialization*, *CreateConnector*, *AddDecomp*, *AddNewAttribute*, *GetAttributeValue*, *GetSpecializations*, and *GetDecomposition*. The

names of these methods are self-explanatory. In essence, the entity object allows a user to perform the full range of functions which are to be expected in an object-oriented, coordination-based system. Thus a user is able to – by a single API call – add a new specialization to an entity, without worrying about inheritance or entity creation. The magic of abstraction in the server takes care of all low-level details.

3.2.2 Relation

Entity objects are related to each other via *relation* objects. Relation objects provide a physical realization of a binary relation existing between two entity objects. There are various types of relations: (1) a *specialization* relation exists between an entity and its specialization, (2) a *decomposition* relation exists between a parent entity and its decomposition child, (3) a *connector* relation exists between two ports which are connected by a connector , (4) a *navigation* relation exists between a navigational node and its navigation child, which may be another navigational node or an activity, (5) an *is_managed_by* relation exists between a dependency and its managing activity, or between a port and its managing port, and (6) a *has_resource* relation exists between a dependency and the resource it manages. Relations may also be inherited or replaced by other relations, in which case a reference is maintained to the relation which is inherited from or replaced.

The relation object's most common properties and methods are *Name*, *ID*, *RelationType*, *StartID*, and *EndID*. The last two methods return the ids of the starting and ending entities of the relation. An example use would be when a client uses the *GetDecomposition* method on an entity to get a collection of relations. Then, the client uses the *EndID* property of each relation to get the entities that exist in the decomposition of the parent entity. Additional properties such as *SlotID* were added to the relation object, and are discussed in section 4.3.3.

3.2.3 Attribute

All entity objects have attributes, which are realized physically in the object server by *attribute* objects. An attribute object contains a reference to the entity of which the object is an attribute. An attribute object also has a unique id, a name, a value, a default value and some information on its inheritance behavior. As mentioned earlier, attributes are inherited down the specialization hierarchy. The information on an attribute's inheritance behavior determines if it will inherit the value of its parent, assume a default value, or not inherit a value at all.

3.2.4 PHDB

The highest level object exported by the API is the *PHDB* (Process Handbook Database) type. This object provides methods to open or close a database, get its root entity, or get an entity object by passing in its id.

3.3 Extensions to the Object Server

To fully support the Dependency Editor design, it was necessary to extend the object server and the database schema to support additional functionality. A brief synopsis of these changes is presented below.

3.3.1 Caching

Since disk access time is usually the bottleneck in database systems, it was necessary to implement some caching between the bottom and middle-tiers (see 3.1) of the Handbook's architecture.

Most of the calls regarding an entity object to the server were found to be those relating to counts of other related entities. More specifically, API calls relating to the number of children in an entity's specialization and decomposition, or the number of bundles underneath it, were found to be the most frequent. The base level server implementation dealt with these API calls in a rather ludicrous fashion; for each

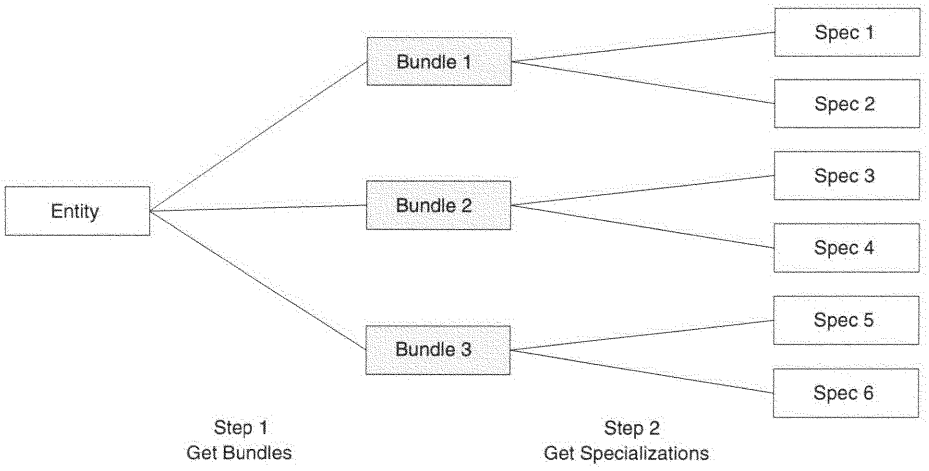

Figure 3-2: Getting the Specializations of an Entity

function call to get the specialization count of an entity, calls were made to initialize every entity in the specialization. Finally, these entities were counted and the final count was passed back to the client. Notice that initializing each entity meant an additional number of disk (slow) accesses. Thus simple count-related calls ended up doing a lot of unnecessary work.

The solution to this timing problem was a traditional systems level approach: the use of caching to speed up each call. The database schema was enhanced to support entries for specialization, decomposition, and bundle counts. These were computed at a low amortized cost each time a new entity was created, and stored as part of the memory-resident entity object itself. Updates to counts – far less frequent than count queries – meant some work in writing to the memory resident object as well as to disk, but system performance increased significantly. A write-through technique was utilized to keep the cached counts fresh: changes were written immediately to disk whenever an update was performed. Although better cache update techniques exist, write-through is particularly simple to implement and maintain.

Another similar performance boost involved changes to the *GetSpecializations* method. In the old database schema, bundles – which are supposed to be attributes in the high-level logic – were stored as specializations of the parent entity. Specializations

of the entity were stored as specializations of the bundles. This caused enormous overhead in terms of database queries. To get the specializations of an entity, the server first performed a query to get all the bundles and then queried each bundle to get all the specializations.

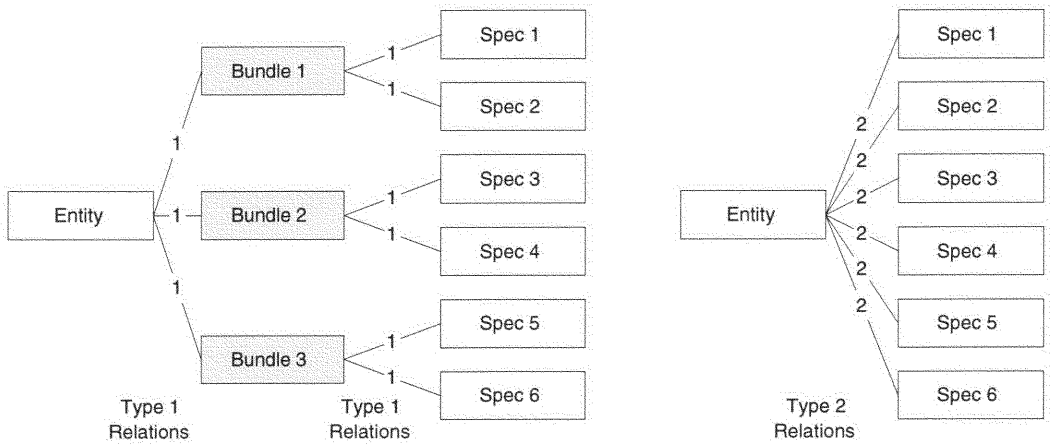

Figure 3-3: Types of Specialization Relations

The fix, as before, was to introduce caching. This required the addition of a new *type* property to specialization relations. Specializations of type 1 are defined to be primary specializations of the sort illustrated in Figure 3-2. The extension was to add specialization relations of type 2, which were cached relations from the parent entity directly to the specializations: this is shown in Figure 3-3.

Note that getting the specializations of an entity now involves just walking down a single step in the type 2 specialization relation tree. Considerable improvement in system performance was observed after this modification to the schema.

3.3.2 Addition of Slots

Another necessary extension to the Process Handbook semantics was that of *slots*. The concept of slots is similar to that of frames and slots from artificial intelligence. Essentially, it means that every entity in the decomposition of another entity is said to occupy a slot in that decomposition.

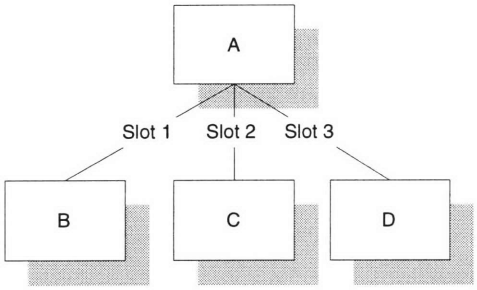

Figure 3-4: Decomposition Slots

Figure 3-4 illustrates these semantics. Associated with each item in the decomposition of A is a slot value. Slots are unique in the context of a single decomposition. A key feature of slots is that they are inherited: any specialization of A will inherit the slots 1, 2 and 3.

As shown in Figure 3-5, the specialization A' of A has inherited the slots of A. Notice that even though the entity D in the decomposition of A has been replaced with D' (a specialization of D) in the decomposition of A', it still has the same slot value. Thus, slots carry more information than just the relation links that they tag. In particular, they present a method to preserve the derivation information in the context of a decomposition when a sub-process undergoes successive refinements. Since the slot ids are preserved, the similarities between the most general process and the final, refined process are apparent from observing the slot values assigned in the decomposition of each process.

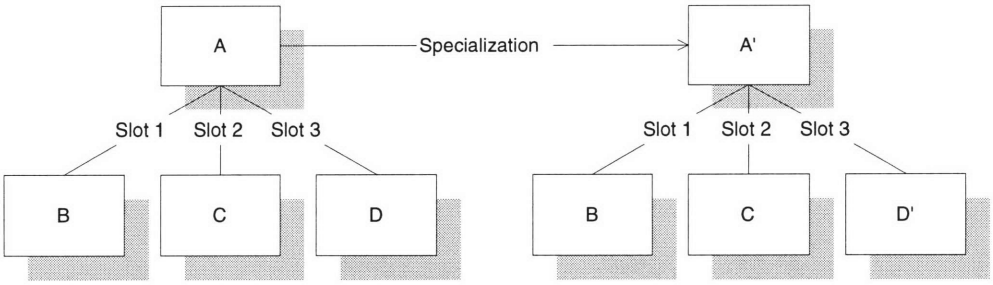

Figure 3-5: Slot Inheritance

One problem with a non slot-based system relates to connector rendition on the client side. Consider the case where an entity exists twice in the decomposition of its parent entity. Any connector connected to a port of this entity is ambiguous in the sense that it is unclear as to which instantiation of the child entity it is connected to. The next section discusses how modified connectors built on the powerful slot idea can resolve this dichotomy.

3.3.3 Modification of Connectors

As pointed out in the last section, dealing with connectors in a decomposition context where a single entity occurs twice is ambiguous. A point worthy of mention is that although the same entity occurs twice in the decomposition and therefore has two instantiations of each of its ports, the path of slots right up to the parent is unique for every entity.

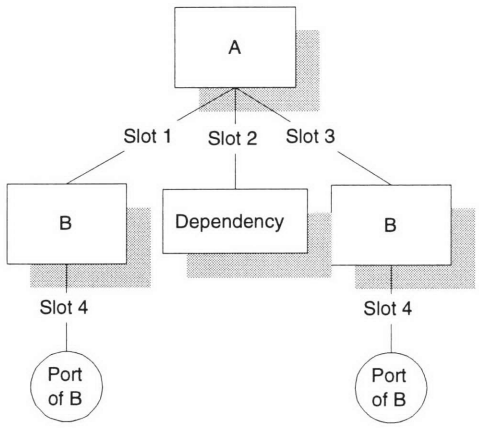

Figure 3-6: Slot Path Uniqueness

Figure 3-6 illustrates this concept; notice that the same B exists twice in the decomposition of A, so the two circles labeled 'Port of B' actually represent the same port. Since B occurs twice in A's decomposition, it occupies two different slots. Even though it is the same entity, it has two different slot values in each of its instantiations. The port of B however, exists in the decomposition of B only once so it has the same slot value in both instantiations. Notice however, that the path of slot ids up to the

final decomposition parent is unique for every instantiation of the port (*Slot 4 – Slot 1* in the first case and *Slot 4 – Slot 3* in the other).

A connector connected to the port of B must be aware of the instantiation to which it is attached. This was the crux of the modification to the Handbook semantics. The database schema was extended to support a *context_path* property for connectors, which stores the path of slots from each of the start and end ports of the connectors right up to the context parent.

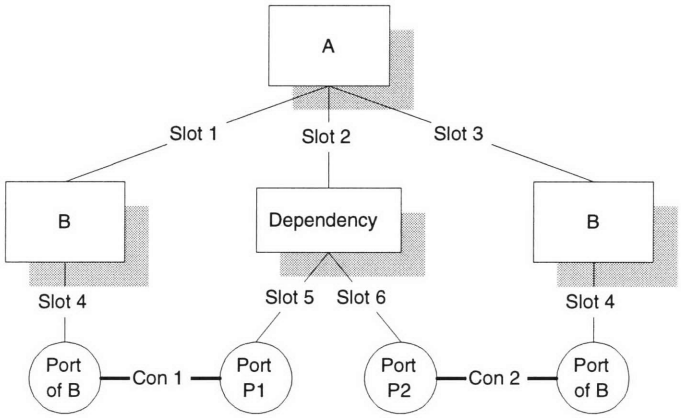

Figure 3-7: Slot Aware Connectors

Figure 3-7 illustrates the modified connectors. The connector labeled 'Con 1' between 'Port of B' and P1 stores the path of slots from each port to A. Thus, included in the connector definition are the two paths *Slot 4 – Slot 1* and *Slot 5 – Slot 2*. Similarly, the connector 'Con 2' is defined by the paths *Slot 4 – Slot 3* and *Slot 6 – Slot 2*. This modification lends tremendous flexibility to the Handbook. A restriction of the old semantics was that every port on an entity was unique in the database, the uniqueness property being necessary to make sense of connector relations. With the new slot-aware connectors, this port uniqueness condition is no longer necessary. Thus, in essence, the same port can be represented many times over in a drawing and all of its connections would be uniquely defined by the context paths of slots.

This extension was invaluable in cutting down port repetitions. It also simplified the inheritance algorithm for connectors. Since the slot property is invariant under

the specialization operation, all new connectors can merely copy over the context slot path information of their parent connector.

3.3.4 Support for Managing Processes

The object server in its base implementation did not provide sufficient functionality for supporting managing processes for dependencies. Some modifications were necessary to introduce the requisite functionality for dependencies in the Handbook.

In the previous version of the object server, dependencies and ports were guaranteed to be unique in the context of a decomposition. This restriction made certain algorithms rather easy to implement. However, in a general context, the restriction does not make sense. In a real-world process scenario, it is perfectly feasible for the same dependency to exist between two processes in two completely dissimilar contexts. This property of the old system also guaranteed the uniqueness of connectors; since each entity was unique in a decomposition, so was a connector.

Coordination science takes the view that a dependency is merely an abstraction for an underlying coordination mechanism – which is said to be the managing process for that dependency – and certain other simpler dependencies. Figure 3-8 illustrates a single-producer, single-consumer dependency which is being managed by a managing process of the correct port configuration. Each port of the dependency is mapped onto a port of the managing process.

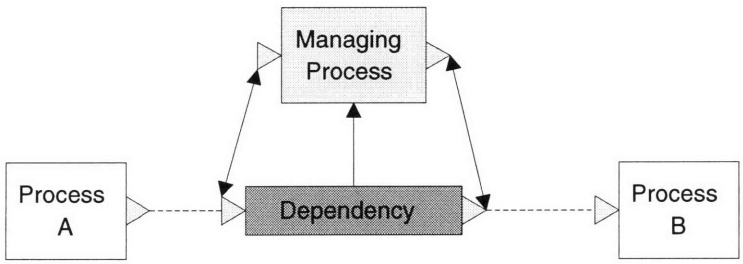

Figure 3-8: Port Matching for a Managed Flow Dependency

Replacing a dependency with its managing activity adds more detail to the high-

level process description. The replaced view includes as many single-producer single-consumer flow dependencies as there were ports on the original dependency. The ports of the new dependencies are connected to the ports of the managing activity and to the ports of the original dependency. Each of these new simple flow dependencies can also be managed by other – possibly simpler – managing processes, and further replacement can give an increasingly refined view of the high-level process.

This model was implementable in the old schema by virtue of the decompositional parent uniqueness property for dependencies and ports. For non-unique ports and dependencies, the following points are worthy of mention.

- The dependency being managed needs to maintain a context-sensitive reference to its managing activity. This condition is necessary because the same dependency could be managed by different managing processes when it exists in different decompositional contexts.

- System generated dependencies, ports and connectors need to maintain the context of the parent dependency together with the main decompositional context of the dependency's parent. Again, since the main dependency may exist in many decompositional contexts, this condition of maintaining a *primary* context (i.e. to the dependency) and a *secondary* context (i.e. to the parent activity) is essential for providing the required functionality.

The implementation of managing activity support for dependencies required a schema change. Namely, system generated entities in the new model also need to keep a reference to their secondary context. The following changes occur when a managing process is set for a dependency.

1. The system introduces an *is_managed_by* relation between the dependency and the managing process. The system also introduces an *automatic_decomposition* relation between the dependency's parent activity and the managing process. An *automatic_decomposition* relation can only be generated by the system, and differs from normal decomposition relations by virtue of the fact that the *Get-*

Decomposition method for entities does not return this type of relation. *Automatic_decomposition* relations are only exported when the object server receieves a request to replace a dependency with its managing activity. The act of introducing an *automatic_decomposition* relation establishes the secondary context of the managing activity to be the parent activity.

2. Assuming that the ports of the dependency are matched with the ports of the managing process, the system creates as many single-consumer single-producer flow dependencies as there were ports on the original dependency. The system also generates the right types of ports for these dependencies. More *automatic_decomposition* relations are added between the main dependency's context parent and the newly created dependencies; this completely establishes the secondary context of the managing process and the synthetic dependencies.

3. Now, the system generates connectors; for each simple flow dependency, the connector connecting one port of the dependency to the port of the managing activity is simple to generate. The context path information of the new connector simply follows the path of slots from the dependency's own port to the secondary context entity, and then follows the slot path of the managing activity's port up to the secondary context entity. The other connector is similar: it connects the other port of the synthetic dependency to the port of the original dependency by constructing slot paths for both end points. This demonstrates the full power of the semantics of having slots in a decomposition.

4. The connectors contain a reference to the high level parent process as their contexts. In addition, each connector also maintains a *managed_by* reference to the dependency which yielded it in the first place. These two pieces of information uniquely specify the connection relation. Notice that the newly created synthetic dependencies can now be replaced by their managing processes. The connectors created by this act will have references to both the high level activity and the dependency which generated them – in this case the simple synthetic dependency. Thus, each connector created by successive refinements is uniquely

defined by the primary and secondary contexts of its yielding dependency as well as the high-level parent process.

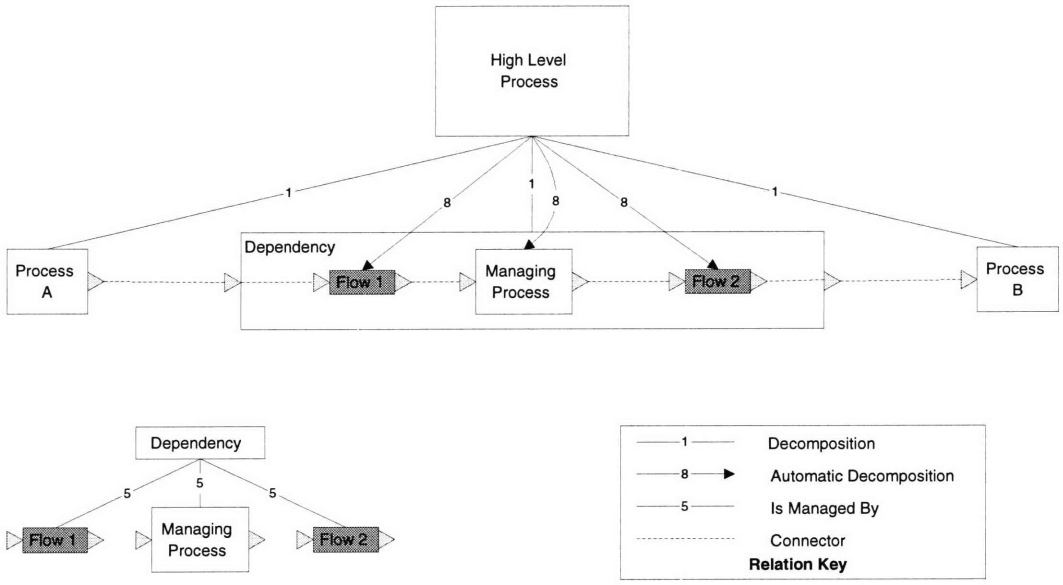

Figure 3-9: Setting the Managing Process for a Dependency

As shown in Figure 3-9, the two dependencies *Flow 1* and *Flow 2* are generated when the ports of *Dependency* are matched with the ports of *Managing Process*. *Automatic_decomposition* and *is_managed_by* relations are created and synthetic connectors are drawn as shown in the frame of *Dependency*.

Chapter 4

The Dependency Editor

4.1 Introduction and Motivation

The Dependency Editor offers a view of process descriptions where dependency relations between activities are emphasized. This view comes at the expense of some information loss in terms of specialization relations and inheritance, which is provided by the Specialization Viewer – another integral part of the Process Handbook system. The Editor provides a robust methodology for viewing, editing, and analyzing dependencies and their managing coordination processes. It enables users to view a process at an arbitrary level of detail, as opposed to other viewers in the Handbook, which impose a hierarchical restriction on views. The Editor allows users to lay out visual process descriptions on a two-dimensional plane, and to explore processes in successively greater levels of decomposition detail. Furthermore, it gives users the additional capability to set managing resources and coordination processes for dependencies, and to replace dependencies by their respective managing activities.

One possible dependency editorial scheme was described in [3]. However, this editor was developed for an older version of the Process Handbook and is now outdated. The current Handbook interface – built on top of an OLE server – has extensive viewing capabilities, but lacks robust support for editing process descriptions. Thus, a flexible yet robust editor was needed for the system to simplify process creation and manipulation. The Dependency Editor fills this void in the current system's

implementation.

4.2 Visual Metaphors

Like any process manipulation tool, the Dependency Editor relies on some core visual metaphors. Visually, the Editor is required to display a variety of different object types, namely activities, dependencies, ports and connectors. The Editor implementation adopts a simple, yet elegant visual representation for each of these types. Activities and dependencies are represented as boxes, which may be open or closed depending on their decomposition state (more on this later). Dependencies are generally rendered to be thinner and more refined than an activity rendition. Of course, the user is provided with easy methods to resize or reshape the visual rendition of an activity or dependency. A flashy color schema is also displayed for dependencies, thus making a dependency rendition stand out with respect to its neighboring objects. Ports are displayed as smaller objects along side their parent activity or dependency. Consumer and producer ports are rendered differently to illustrate their different functions. Untyped or generic ports are rendered as simple square objects, signifying that the user can further type this particular port into either a consumer or producer. Connectors – both horizontal and vertical – are displayed as lines of unity width.

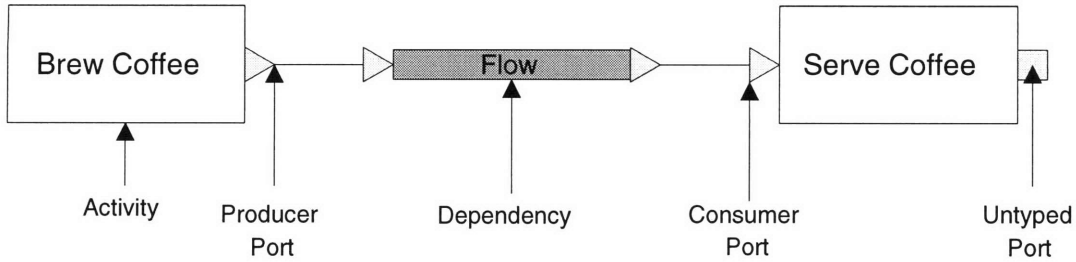

Figure 4-1: Visual Representation of Various Types

One major design decision about the implementation of the Editor relates to the

dimension of decomposition. Earlier implementations of dependency viewers relied upon a *click-and-replace* metaphor when dealing with decompositions. Simply put, users could view sub-processes in a process description by clicking on the process's visual representation. The system would then respond by hiding the current process display, and replacing it with renditions of the sub-processes in the parent process's decomposition. This metaphor had the advantage of reducing screen clutter. The total amount of information presented to a user at any point of a process viewing session was bounded by the maximum number of processes existing at any level of the decomposition hierarchy. Thus, the user was served with a simplified representation. For older machine architectures, this rendition was useful because operating systems like Windows 3.1 placed a low bound on the number of objects that could be displayed at one time. This particular metaphor was therefore light on system resources.

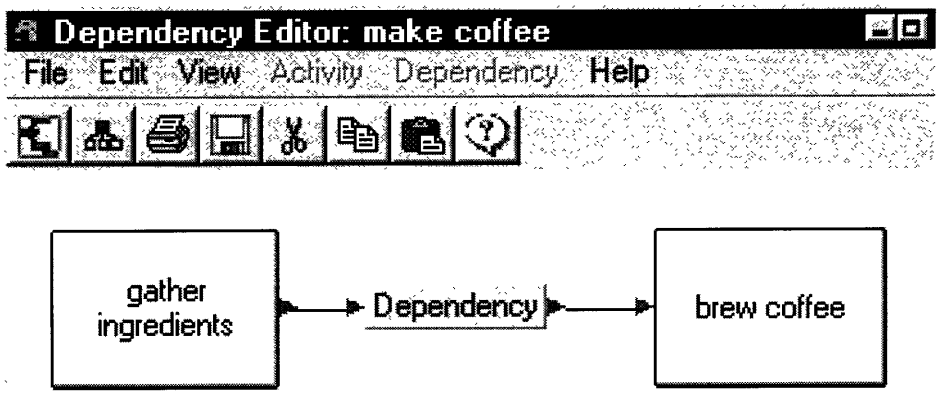

Figure 4-2: *Click-and-Replace* Representation of *Make Coffee*

A major downside of the click-and-replace metaphor is that it is unable to render the semantically useful metaphor of a vertical connector. As pointed out in Section 2.2.3, vertical connectors are an essential part of the Handbook's semantics. Dependencies deep within a decomposition rely on the semantics of vertical connectors to form their definitions at higher levels of the decomposition. Another con of this metaphor is that it poses significant information loss at every level of decomposition. In most cases, a user would prefer to preserve context information as to how the

current process viewing session arrived at its present state. Since this metaphor can only show an activity or its decomposition parent at the same time, no context information is presented to the end user. Figure 4-2 shows these drawbacks of a complete replacement strategy in a decomposition context. Firstly, even though vertical connectors exist between the ports of *make coffee* and its decomposition children, there is no way to render them since *make coffee* is not visible. Secondly, simple observation of the decomposition children gives no indication of the decomposition context; i.e. the user receives no visual cues about the parentage of *gather ingredients* and *brew coffee.*

The visual metaphor for decomposition presented in the Dependency Editor is dubbed the *worlds-within-worlds* model. Essentially, this means that all context information in a decomposition hierarchy is preserved while navigating through different levels of a process description. When a user wishes to view the sub-processes in a process's decomposition, the system opens the process up in the form of a frame. Sub-processes are rendered in this frame such that both a sub-process and its decomposition parent are visible at the same time. This model can then be applied recursively to the sub-processes until a sufficiently detailed view of the process is presented to the user.

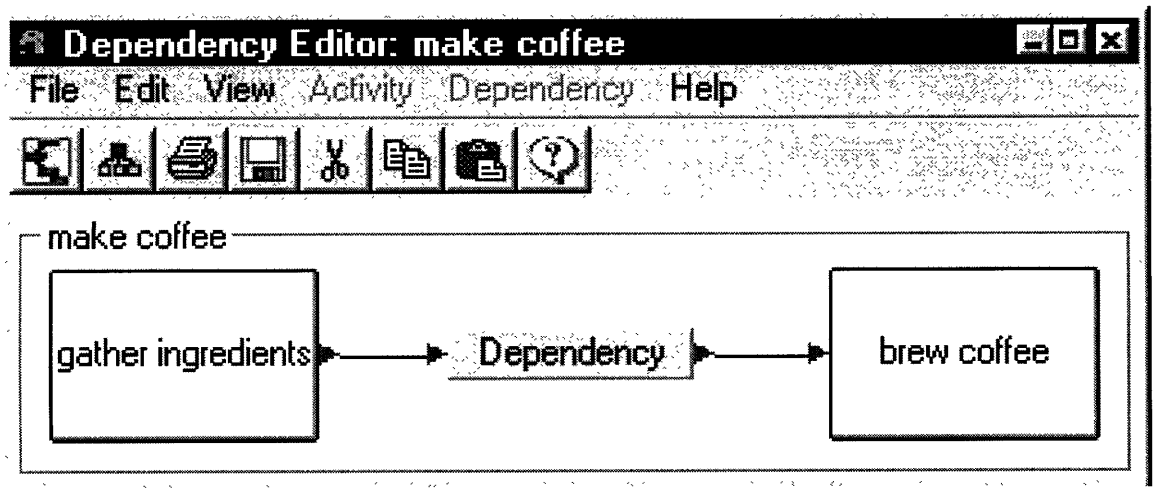

Figure 4-3: *Worlds-within-Worlds* Representation of *Make Coffee*

A major pro of the worlds-within-worlds view is that vertical connectors can now be rendered and made to illustrate their full semantic content. Also, since arbitrarily many decomposition levels can be visible at the same time, there is no information loss from the user's perspective as opposed to that experienced with the click-and-replace model. This implementation was made possible with recent advances in operating systems and machine architectures. Windows 95 – for which this system is designed – running on an average Pentium processor can easily render more objects than can be realized even in very detailed process descriptions. Thus, the use of this model makes much more sense. Also, since sub-processes are rendered inside the frame of their decomposition parent, a strict abstraction is maintained, in which different levels of a decomposition hierarchy are cleanly separated via frames at each level, and interact only by means of a series of horizontal and vertical connectors. Figure 4-3 illustrates the *make coffee* process in the worlds-within-worlds view. Notice that vertical connectors are visible and that decomposition context is maintained since *make coffee* is visible at the same time as its decomposition children. Also, *gather ingredients* and *brew coffee* can be expanded further while still preserving the viewer's decomposition context.

4.3 Base Implementation

Some discussion of the implementation of the Dependency Editor is now merited. The Editor is an efficient tool for viewing, editing, and analyzing processes, their related dependency relations, and the underlying coordination mechanisms which manage these dependency relations. Section 4.3.1 will discuss how visual representations of each of the major object types of the Process Handbook – namely activities, dependencies, ports and connectors – are implemented. Sections 4.3.2 and 4.3.3 will discuss how these objects are positioned according to the underlying database-enforced positioning and how they interact.

4.3.1 Object Types and Methods

The Editor utilizes a bijective map between Process Handbook object types and visual renditions of these types. For every object type in the Handbook, there exists a higher level type of the same name beneath the graphical user interface of the system. These higher level types include the Handbook object as part of their description. In addition, they also contain model information about their visual rendition, positioning and connections with other objects of similar or different types.

1. Activities

   An activity in the object server is a generic entity object of type activity. From the Editor's perspective, an activity is realized in the form of a higher level *DE_Activity* (Dependency Editor Activity) type. Some of the more relevant methods and properties of *DE_Activity* are as follows.

   - The *PHDB_Entity* property contains a reference to the actual Entity object exposed by the server.
   - The *ParentRelation* property contains a reference to the decomposition relation which initially yielded this activity.
   - The *Display* property references the object's display, which may be a 'closed' box – in case of an un-expanded activity – or an 'open' frame in the case of an expanded activity, where the decomposition children are visible in the frame.
   - The *CreateExisting* method uses a server relation object to create a visual *DE_Activity* object representing the end activity of the relation. It also sets the appropriate display, entity and relational references.
   - The *CreateNew* method creates a new entity in the Process Handbook database according to user input and initializes a visual *DE_Activity* object for the new entity.
   - The *Expand* method queries the object server to get other objects in the decomposition of the current activity. The method then creates appropriate

visual activities, dependencies and ports for the objects in the decomposition. Finally, the method creates a visual connector between visual ports for each connector relation in the decomposition. An interesting algorithm is utilized for the connector rendition scheme; each connector contains information about paths of slot ids from the start and end ports, right up till the context decomposition parent (for a detailed discussion of slots and connectors, please see section 3.3.2). The algorithm locates each port of a connector by first deciding if any of the activity's own ports match the slot path definition of either of the connector's ports. If no match is found, then the algorithm performs the same test for the ports of each of the other objects in the activity's decomposition. Finally, when both ports have been found, a visual connector is drawn between them.

- The *Connectors* property maintains a collection of all the connectors connected to this object.
- The *Ports* property maintains a collection of all the visual port objects that are attached to this object.
- The *PositionPorts* method positions ports on the activity's surface according to the method's arguments.

Other properties and methods maintain references to the form which contains the visual representation of the activity, the visual parent object of this activity, and abstract away useful functions like adding entities to the decomposition of the entity.

2. Dependencies

A dependency in the object server is a generic entity object of type dependency. From the Editor's perspective, a dependency is realized in the form of a higher level *DE_Dependency* type. Currently, this type has properties and methods which are virtually identical to those of the *DE_Activity* type. A discussion of these methods has already been presented in the previous section.

3. Ports

   A port in the object server is a generic entity object of type port. From the Editor's perspective, a port is realized in the form of a higher level *DE_Port* type. A majority of the properties and methods of the port type dealing with references to the actual port entity and visual display are similar to those presented for activities. A brief discussion of some of the different methods and properties is given below.

   - The *Display* property is identical to the display property for activities. However – as mentioned before – the display of a port depends on its type.
   - The *PortType* property contains information on the type of the port e.g. a producer port.
   - The *Connectors* property maintains a collection of all the horizontal and vertical connectors which are attached to this port.
   - The *AlignConnectors* method aligns all connectors connected to this port after a typical move or create operation.

4. Connectors

   Connectors are relation objects in the Handbook of type connector. A connector is fundamentally the most different type of object rendered by the Editor since it signifies a binary relation between objects and not a single object. A connector is represented by the higher order type of *DE_Connector*. Some of the more relevant properties and methods of the *DE_Connector* type are as follows.

   - The *Display* property – as discussed earlier – maintains a reference to the connector's visual display (a simple line in this case).
   - The *StartEntity*, *EndEntity* and *InContextOf* properties reference the connector's start, end and context objects, respectively.
   - The *CreateExisting* method creates a visual connector from the information in an actual Handbook connector relation.

- The *CreateNew* method is slightly more interesting; it is invoked when a user tries to create a connector between two ports on the display. The method builds up paths of slot ids (see section 3.3.3) from the start and end ports and passes these to the object server's connector creation method. Finally, a new visual connector object is created.

The remaining properties and methods of the connector object are similar to those for activities mentioned earlier. These maintain the proper references to the actual connector relation, the visual connector's containment form, its parent object. etc.

4.3.2 Screen Layout

Since the Editor is supposed to be a robust process analysis tool, it is necessary for it to provide a mechanism for storing layout information. This is done by saving layout information as an attribute of the relevant Handbook's entity object. Each entity stores layout information about its size, the relative positioning of its decomposition children within its frame, and the positions of its ports. This information is stored as a string in the *Positions* attribute of the entity.

A call to the *Expand* method of the visual object replaces the closed box display with an open frame, gets the decomposition children of the entity, gets the value of the *Positions* attribute as a string, parses this string, and finally moves the decomposition children around according to the value of the *Positions* attribute. Note that the *Positions* attribute is relevant only for activities and dependencies, since ports are defined only as surface structures of these types, while connector positioning is fully defined by the positioning of the connected ports.

4.3.3 Interaction of Objects

The above mentioned activity, dependency, port and connector objects interact in a number of ways to produce elegant process representations. Methods for these types abstract away process editing functions such as adding to a decomposition of an ac-

tivity, viewing the decomposition of a process, creating connections between processes and dependencies, creating dependencies between processes, etc. The various links between these object types are now briefly discussed.

Activity objects contain other *activity*, *dependency* and *connector* objects in their decomposition frame. This frame rendition shows the underlying decomposition relations – exposed by the Process Handbook object server – between the activity and its decomposition children.

As per the Handbook semantics, *dependency* objects can only contain other *dependency* objects in their decomposition frame. The Handbook also allows ports to exist in the decomposition of a dependency, but ports are not visually rendered in a decomposition frame.

Activity and *dependency* objects contain *port* objects as part of their definitions. In the underlying Process Handbook semantics, ports are actually part of the decomposition of an entity. However, decomposition relations are not useful for the purposes of visual rendition. Thus, port objects are displayed as smaller structures on the surface of their parent object's display and not as decomposition children.

Connector objects connect port objects in a decomposition frame. This outlines the underlying semantic of connector relations existing between port entities in the context of a decomposition.

On a higher level of visual abstraction, dependencies exist between *activity* objects. These are realized by a *dependency* object, its *ports*, the ports of the activities, and the *connector* objects between the ports.

The type of a *port* object defaults its position on the surface of an *activity* or *dependency* object, illustrating a general left-to-right flow on the overall diagram. Thus, a producer port on an activity shows up on the right side of the activity while a consumer port shows up on the left. If the ports are connected – via a dependency and two connectors – then the general feel of the process is that of a left to right flow. Of course, the user can overrule this default behavior and position the port on any part of an activity or dependency object.

4.4 User Interface Level Functionality

The Dependency Editor provides users with full viewing and editing capabilities for process manipulation. The Editor receives user input by a series of keyboard and mouse clicks. Each user action is interpreted in a natural way to provide a powerful and robust environment for process analysis.

4.4.1 Getting the Decomposition of an Entity

In the context of viewing processes at varying levels of detail, the dimension of process decomposition is invaluable. Since this is the most commonly used function in a process viewing exercise, it was necessary to make the expand operation on an object as user-friendly as possible. Thus, a mouse double-click on a visual activity object returns the decomposition of that activity. When this action is performed, the *Expand* method of the underlying object is called which replaces the closed box display of the activity by an open frame display and renders each of the decomposition children within this frame, positioned according to the database information on this particular activity. Finally, visual connectors are drawn between the relevant ports of the entity or its decomposition children

4.4.2 Creating Connectors

Creating connectors between ports in a visual display is also a common operation during a process creation or editing session. Thus, the Editor provides a particularly simple user methodology for this task. To create a connector between two ports, the user begins by clicking on the start port of a connector. The system displays a connection line rooted at one end to the start port, and varying on the other end as the user moves the mouse pointer over the display. Finally, the user can create a connector by clicking on the desired end port of the new connector. At this point, the system performs a compatibility test between the two ports. If the ports are found to be incompatible – for example, if both are ports on dependencies – then the system generates a helpful error message. Otherwise, a connector is created between the two

ports.

4.4.3 Creating Dependencies between Activities

More often then not, a user will be interested in drawing a higher level dependency between activities rather than the lower level connector relations. Thus, the system provides a particularly useful dependency drawing mechanism. Essentially, the user follows the same procedure as outlined for drawing connectors. If the system notices that both ports of the intended connector are related to activities, then the user is prompted to confirm that the intended operation is actually a dependency creation. If the answer is yes, then the system introduces a single producer, single consumer dependency in the decomposition frame, and automatically draws connectors from the ports of the dependency to the correct ports on the corresponding activities.

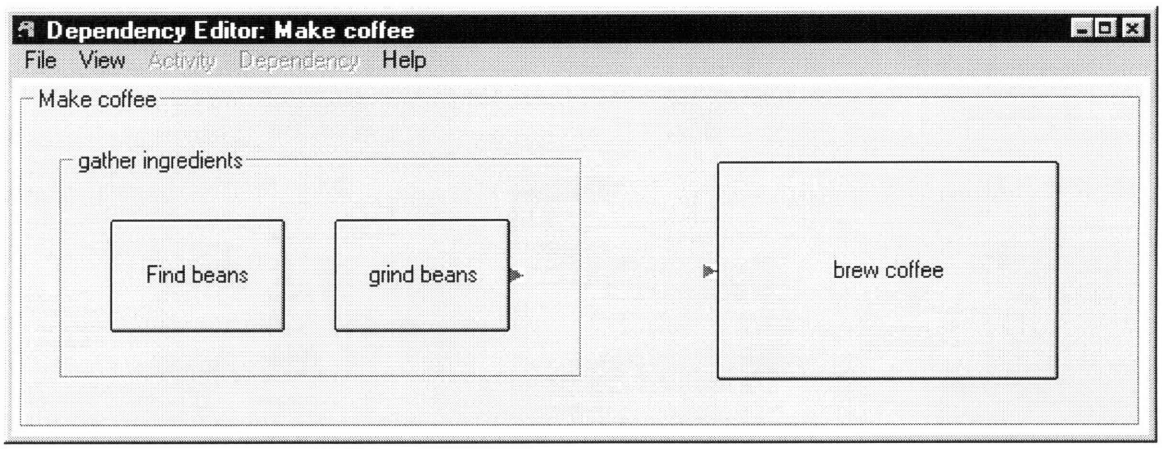

Figure 4-4: Creating a Dependency at Arbitrary Levels in a Decomposition

This operation is slightly more complex when the start and end ports are on activities which are not immediate siblings in a decomposition. In this case, the system responds by constructing decomposition relation paths for each of the start and end activities until the closest common decomposition ancestor of the two is found. Then, ports of the correct type are added to each activity on each decomposition path, and vertical connectors are created between the newly created ports. Finally, a newly created dependency is added to the decomposition of the closest common ancestor

activity and its ports are connected to the relevant ports of sibling activities in the immediate decomposition of the closest common ancestor. Figure 4-4 illustrates this algorithm.

Consider a simple decomposition of the *Make Coffee* process. Suppose a user wishes to create a simple flow dependency between the *Grind Beans* process in the decomposition of *Gather Ingredients* and the higher level *Brew Coffee* process. However, the decomposition of *Brew Coffee* has not been defined yet. In an intuitive sense, this dependency signifies a relation between the high level *Gather Ingredients* and *Brew Coffee* processes. Figure 4-5 shows the result of this operation.

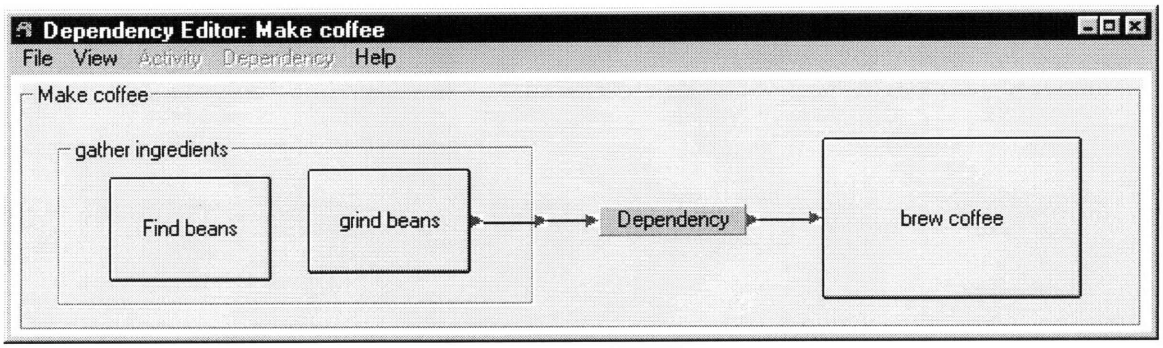

Figure 4-5: Dependency at an Arbitrary Level in a Decomposition

The system first locates the closest common ancestor of *Grind Beans* and *Brew Coffee* – which in this case is *Make Coffee* itself. Then, the system attaches ports of the correct type to processes on the decomposition path all the way up to the closest common ancestor activity and connects these ports with vertical connectors. The only process on the decomposition paths – other then the start and end nodes – in this case is the *Gather Ingredients* process. Finally, the system creates a dependency in the decomposition of the closest common ancestor activity (*Make Coffee*) and attaches the ports of the dependency to the ports of the sibling activities at the top of the two decomposition paths, i.e. *Gather Ingredients* and *Brew Coffee*.

4.4.4 Drag-Drop Operations

The Editor supports drag-drop operations from any other part of the system to the current process editing session. A user can drag an object from any other dependency, specialization, or decomposition viewer and drop it onto the current process's decomposition context. This presents an excellent methodology to create enhanced process descriptions in a bottom-up hierarchical fashion, where existing processes are used as building blocks for other, more advanced processes. Once a user successfully defines a particular process, it becomes a black box which can be used in the decomposition of other processes without explicit reference to its inner implementation.

4.4.5 User Toolbox

The Dependency Editor also includes a toolbox of commonly used objects in the Process Handbook. At any point during a process editing session, the user has the option to click and drag any of the existing processes in the toolbox and add them to the current decomposition. Common toolbox items include generic activities, dependencies, ports, dependencies of varying port configurations, consumer and producer ports, etc. The user can also drag an often used process from a specialization tree onto the toolbox, where it is immediately displayed as a usable object. Moving the mouse over a toolbox item displays a label with the item name.

4.4.6 Setting the Managing Resource for Dependency

A dependency relation usually mediates the flow of a resource between the ports of two activities. Thus, setting the resource for a dependency is a fairly routine operation in the context of the Dependency Editor. A user can set the managing resource for a dependency by clicking on the visual dependency and choosing the 'Set managing resource' item from the pop-up menu. This well result in a modal specialization viewer, displaying all the existing resources in the Process Handbook database. The user can then select an existing resource or create a new specialization to be the managing resource for the dependency.

4.4.7 Setting a Managing Activity for a Dependency

A dependency relation is managed by a coordination process. In fact a dependency can be said to be a higher level visualization of the coordination process and other simpler dependencies; thus a dependency is little more than an abstraction to hide the detail of the underlying managing coordination process. Thus, setting the managing process for a dependency is by far the most useful function of the Editor from a coordination theoretic perspective.

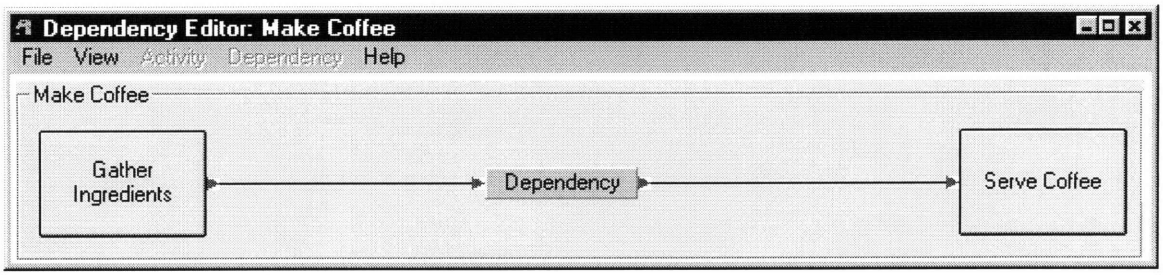

Figure 4-6: Setting the Managing Activity for a Dependency

A user can set the managing activity for a dependency by following a procedure analogous to that for setting the managing resource. The Process Handbook database contains an extensive library of managing processes, and the user can either select a process from this library or create a new one depending on the particular use of the process. The only caveat to this process is that the user must also specify a bijective mapping between the ports of the dependency and the ports of the managing activity, in the case where such a mapping is not immediately obvious. In the simple case of a single-producer or single-consumer dependency, corresponding ports are matched automatically by the system.

4.4.8 Replacing a Dependency with its Managing Activity

Once the managing process for a dependency has been set, the user can choose to replace the dependency with the managing activity to provide a more detailed view of the overall process. The user can perform this action by choosing the appropriate

item from the dependency's pop-up menu.

This results in the visual creation of the managing process and as many single-producer single-consumer dependencies as there were ports on the original dependency. Each of the resulting simple dependencies can then be managed by other managing activities. Thus, this replacement procedure can be carried out recursively on the simpler dependencies until a suitably detailed view of the overall process is obtained.

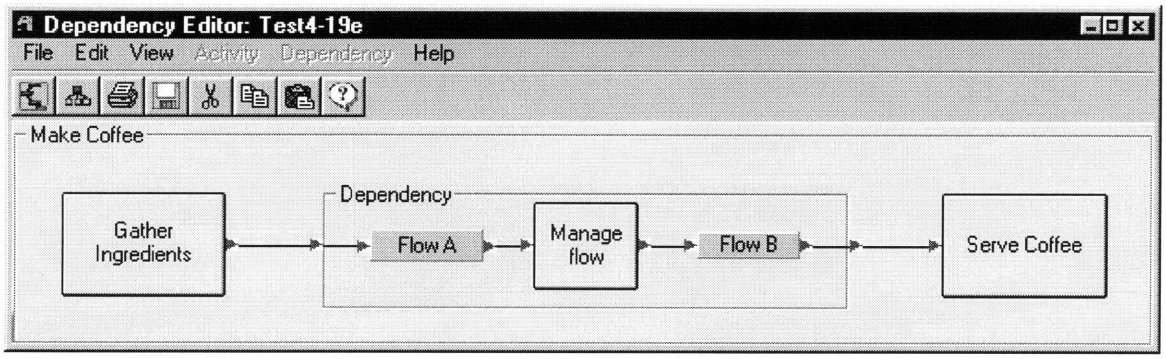

Figure 4-7: Replacing a Dependency with its Managing Activity

Figure 4-7 illustrates this function. The simple flow dependency in Figure 4-6 has been replaced with its managing activity called *Manage flow*. This resulted in the creation of two simple flow dependencies *Flow A* and *Flow B*, which can now be managed by other coordination processes to reveal a greater detail of the high level *Make Coffee* process. The parent dependency is also visible; the user has the option of viewing either the high-level dependency itself or its replaced version in which further detail about the underlying managing process is visible.

4.4.9 Replacing an Activity with its Specialization

Another common procedure in a decomposition context is to replace an activity by one of its specializations. The user can perform this task by choosing the appropriate item from the pop-up menu for the activity. This results in the display of a specialization tree rooted at the current activity, and the user can now either select an existing

specialization, or create a new one to replace the current activity in its parent's decomposition.

4.5 Integration with the System

One major goal of this thesis project was to provide a powerful process editing tool which could be fully integrated with the rest of the Process Handbook. The Dependency Editor achieves this goal by demonstrating full functional integration with the rest of the system.

The Editor supports Windows style drag-drops of items from any open specialization or decomposition viewer into an editing window. A drop onto an open activity frame implies the addition of a decomposition relation between the frame and the drag item. Although the underlying API call may be the same, the actual drop event dispatches on the type of the dropped item. Ports – for example – are not added to the decomposition frame, even though they are added to the decomposition in the strict semantics of the Handbook. The dispatch may also result in error, in which a helpful message is returned to the user. For example, if the user tries to drop an activity object onto a dependency then an error is raised since a dependency can only contain other dependencies in its decomposition.

The Editor also supports drag-drop events from itself to any other viewer of the Handbook. For example, a user can drag an item from an editor window and drop it onto a specialization window. This would result in the object being added to the specialization hierarchy under the appropriate parent. Once again, if the drop event is semantically incorrect, then no action takes place and the user is informed via a helpful error message. *Cut*, *Copy* and *Paste* functions – in traditional Windows style – are also provided. The semantic meaning of a copy and paste is similar to that for a drag-drop operation.

Another useful integration feature is that a details viewer can be raised from any object in an Editor window. The details viewer for an entity lists the attributes of that entity, and provides navigational menus to its decomposition children and

parents, and to its specializations and generalizations. Similarly, the user can raise a specialization or decomposition tree rooted at the current entity from within an Editor Window.

Chapter 5

Using the Dependency Editor

The best way to get the full flavor of the Dependency Editor's functionality is to walk through an actual process modeling example, which makes use of some of the Editor's essential features. The modeling example presented here is adapted from [10].

5.1 A Process Modeling Example

5.1.1 The *Replenish Inventory* Process

Inventory replenishment is triggered at the Retailer based on a balance between sales volume and inventory. An order from a customer may generate a request from the Retailer to the Distributor to supply a quantity of product on a given date. The Retailer next starts a sub-process with the Accounts department to prepare a payment for the Distributor. Inventory is updated and payment is released when the goods have been received and checked.

Figure 5-1 illustrates the current structure of the modeled *Replenish Inventory* process. This process contains eight activities (e.g. *Receive Order*, etc.). The decision represents a conditional flow of activities based on an evaluation of current sales volume and inventory given the retail order details. *Request Prepare Payment* and *Send Order* may be executed in parallel. The *Take Delivery* activity is actually a composite activity which is further defined.

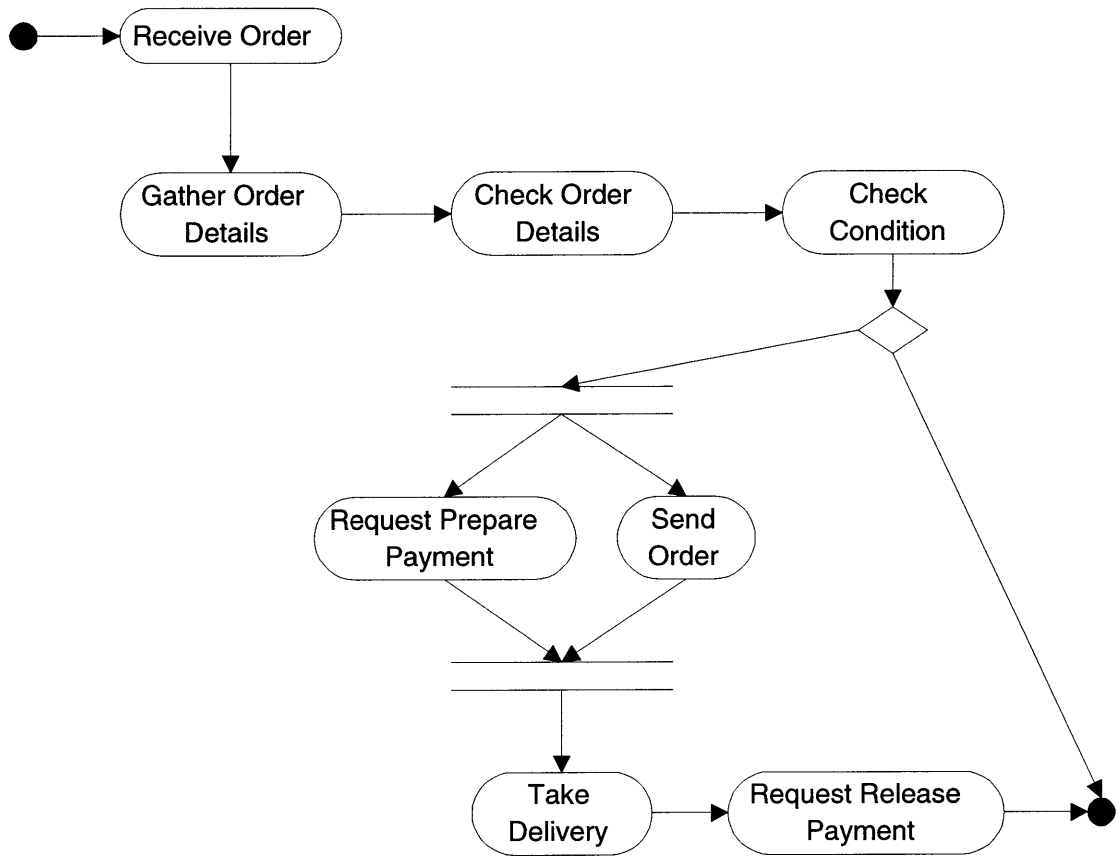

Figure 5-1: *Replenish Inventory (Retailer)* Diagram

The process of taking delivery can be further detailed by considering the sub-activities which are executed during its enactment. Inventory is updated at the retailer when the goods have been received. The delivery must be verified first though in order to ensure that it properly meets the requirements of the order.

Take Delivery is modeled as a simple three-step process in which the goods are received and checked and inventory updated.

5.1.2 The *Replenish Inventory* Process in the Process Handbook

We now consider creating the above process in the Process Handbook, making use of the Dependency Editor.

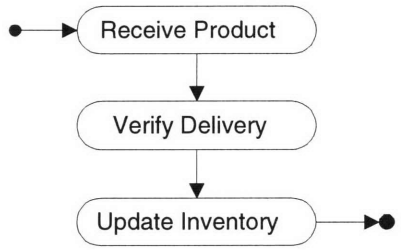

Figure 5-2: *Take Delivery (Retailer)* Diagram

1. Start up the system, and create *Replenish Inventory* as a specialization of a truly generic process.

2. Create a generic single-producer single-consumer activity called *Base* and set it to be the default activity; all sub-activities in the decomposition of *Replenish Inventory* will be specializations of this activity.

3. Open the Dependency Editor rooted at *Replenish Inventory.*

4. Drag the *Base Activity* from the user toolbox into the *Replenish Inventory* de-

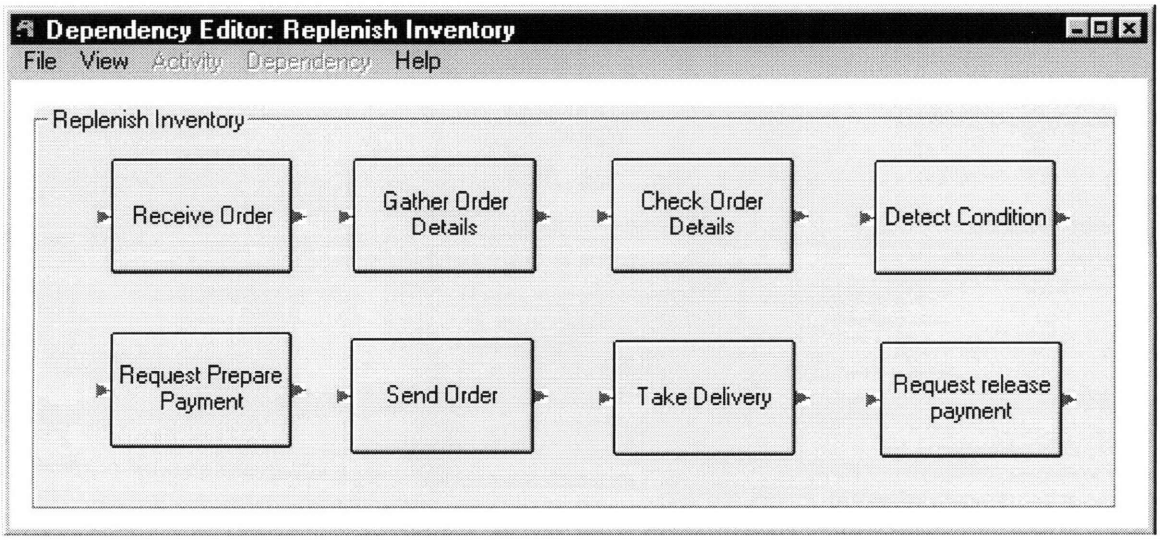

Figure 5-3: Sub-processes in the Decomposition of *Replenish Inventory*

composition frame once for all the sub-activities in the decomposition. This

creates specializations of the base activity; name each sub-activity appropriately.

5. Connect all single start-end port connections; this will automatically generate single-producer single-consumer dependencies.

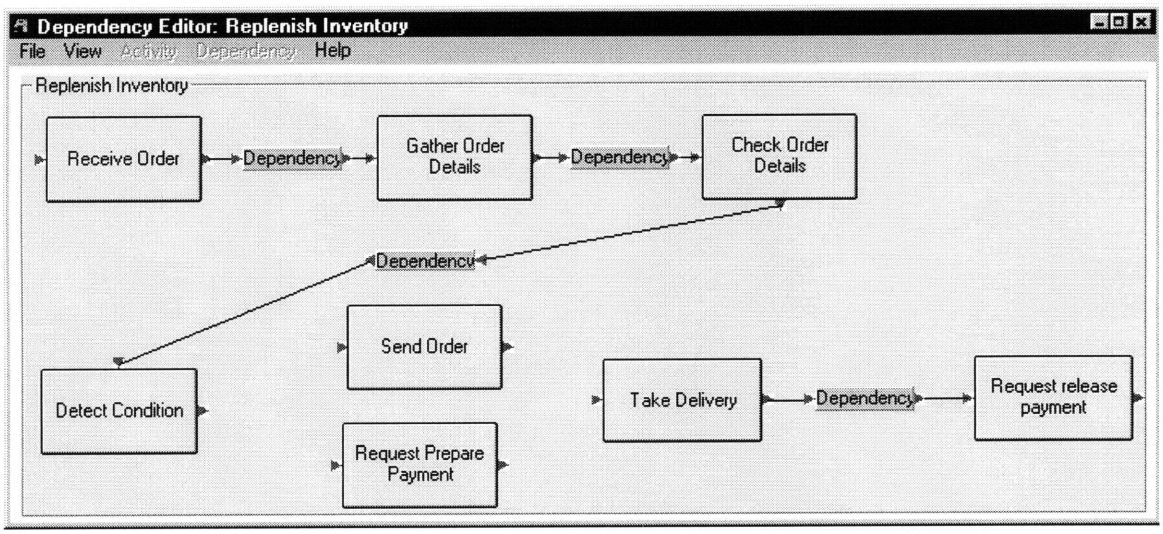

Figure 5-4: Single Producer-Consumer Connections in *Replenish Inventory*

6. Drag the appropriate port configuration dependencies from the user toolbox; draw connectors between the relevant ports.

7. Double-click on the *Take Delivery* process to get its decomposition frame. Drag three instances of the *Base Activity* onto the *Take Delivery* frame, and rename each instance accordingly.

8. Draw the appropriate dependencies between the sub-processes of *Take Delivery*.

This completes the specifications of the *Replenish Inventory* process. Notice that each of the sub-processes in the decomposition of *Replenish Inventory* may itself be a complex composite process. For simplicity, only the decomposition detail of the *Take Delivery* process is illustrated in this example. Again – for the purpose of maintaining simplicity – a one level decomposition view was presented. In practice, the

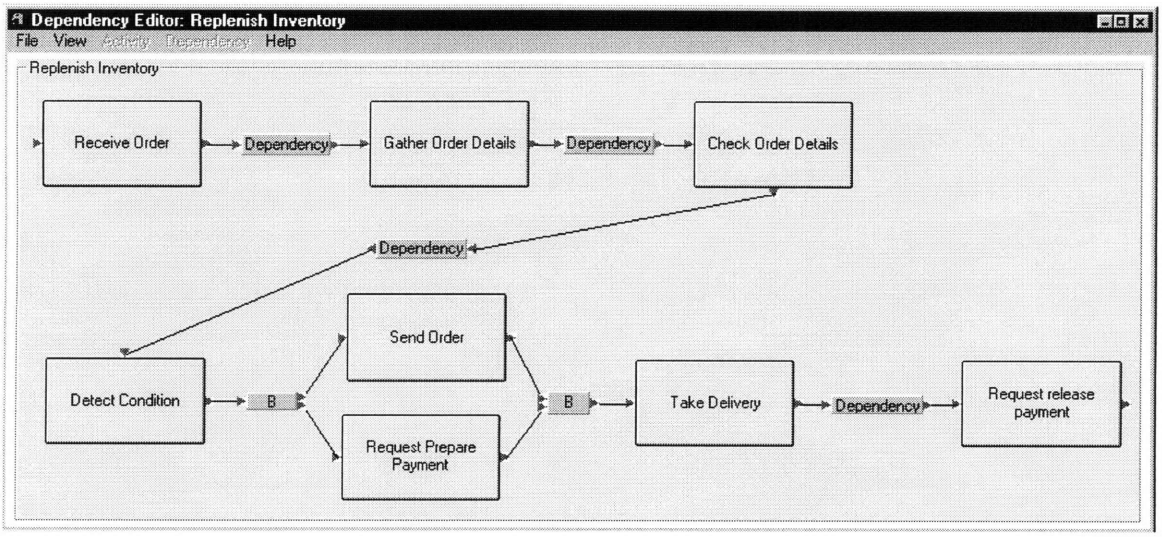

Figure 5-5: A Complete View of *Replenish Inventory*

sub-processes in the decomposition of *Take Delivery* may also be decomposable into simpler processes. The Dependency Editor provides numerous other useful ways to enhance this process description. For example, any of the numerous dependencies in the context of *Replenish Inventory* can be managed by a number of existing managing coordination processes in the Handbook. Replacing the dependency with its coordination process will yield another level of detail – and complexity – to the top-level process description.

An interesting point in this construction is the process design methodology which was adopted. We started by specifying a high-level goal – that of replenishing inventory. We then proceeded by specifying the sub-tasks needed for this high level goal, and then by enumerating the processes in the decomposition of these sub-processes. This design approach is often dubbed as the *Top-Down* method of process engineering.

The opposite approach could also have been taken; namely, we could have started by specifying the primitive processes (like *Receive Product*) first and then successively adding decompositions to achieve the final high level goal of replenishing inventory. This alternate approach is called the *Bottom-Up* model of process design. The Dependency Editor provides robust support for both methodologies: in certain cases

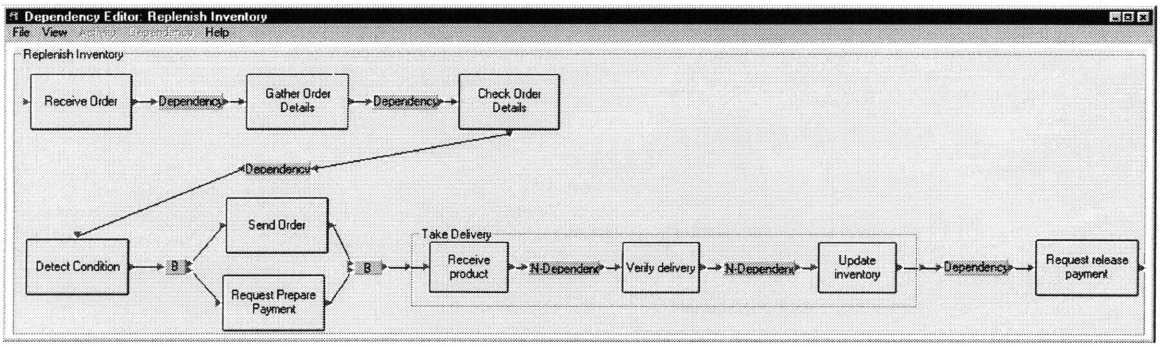

Figure 5-6: Complete Specifications of the *Replenish Inventory* Process

however, users may favor one technique over the other.

5.2 Enhancing an Existing Process Description

We now consider an enhancement of the *Take Delivery* process. This enhancement has three specific goals: (1) specify a detailed description of the sub-activities in the decomposition of *Take Delivery* by adding activities to their decomposition, (2) create dependencies between these newly created activities if necessary, and (3) manage the high-level dependencies by existing managing processes in the Handbook.

Consider the *Take Delivery* process as it currently stands:

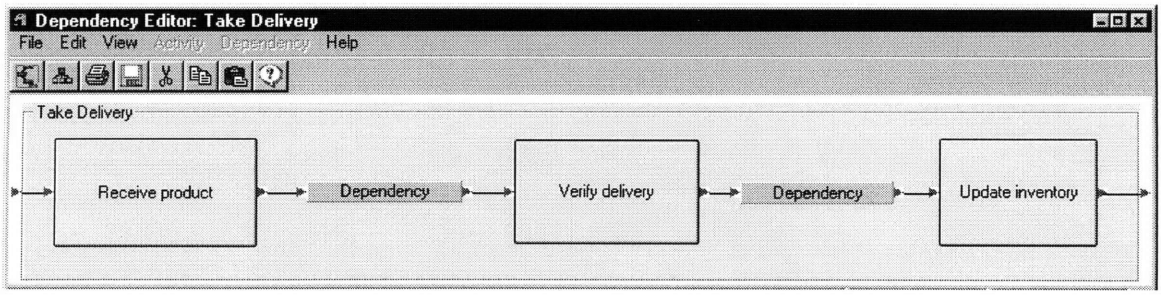

Figure 5-7: The *Take Delivery* Process

1. We wish to enhance this process by adding sub-activities to the decomposition processes. This is done as shown in Section 5.1: *Receive product* is decomposed

into the simpler activities of *Sign voucher* and *Store product.* When receiving a product, the receiver must (1) sign a voucher attested to by the deliverer, and (2) specify storage space for the product. *Verify delivery* is further subdivided into *Check receipt log* and *Call sender.* To verify the delivery, the receiver checks past receipt logs to eliminate errors and then calls the sender to confirm that the delivery is valid. *Update inventory* is extended to two steps *Write data to local database* and *Update warehouse datamart*. Updating inventory involves updating the local database; at the end of the day the data warehouse must also reflect this change.

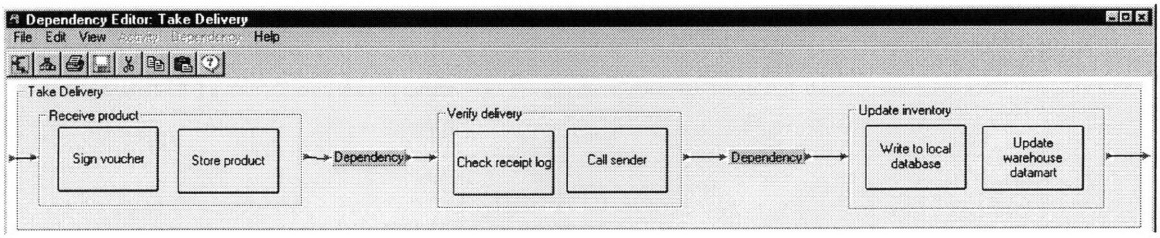

Figure 5-8: Enhancing the *Take Delivery* Process

2. At this stage of the specification process, we realize that there exists a simple flow dependency between storing the product and calling the sender to verify delivery; this dependency is distinct from the high-level dependency between *Receive product* and *Verify delivery.* On a lower level of detail, the high-level dependency may actually be a dependency between *Sign voucher* and *Check receipt log* but at this stage of the process description, exact specifications are not formulated.

   Figure 5-9 illustrates these changes. When the ports of *Store product* and *Call sender* are connected, the system responds by attaching the requisite types of ports to the decomposition parents – *Receive product* and *Verify delivery* in this case. Vertical connectors are added and a new high-level dependency is introduced in the context of the closest common decomposition ancestor – which is *Take Delivery* in this case.

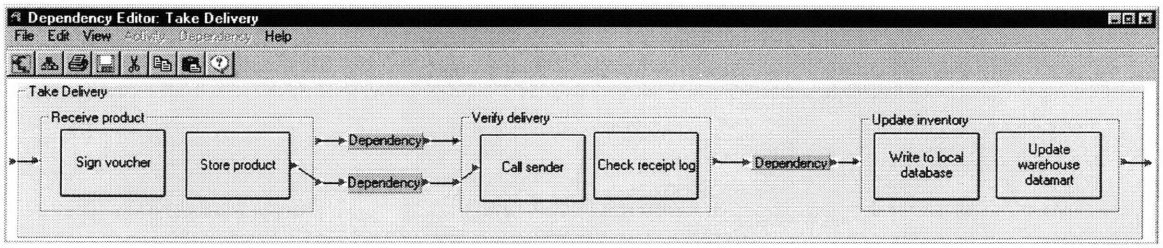

Figure 5-9: Adding a Dependency

3. To proceed with the process specification, simple dependencies are added between sub-activities as shown in Figure 5-10.

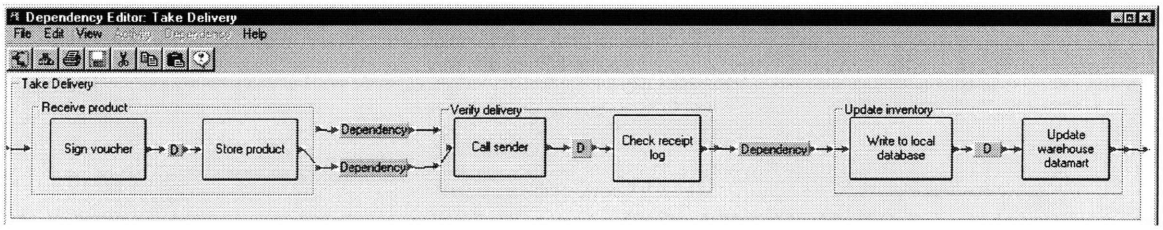

Figure 5-10: Simple Dependencies

4. The dependency between *Verify delivery* and *Update inventory* is now set to be managed by the coordination process *Manage Dependency – Flow.* Figure 5-11 illustrates this change when the dependency is replaced with its managing activity.

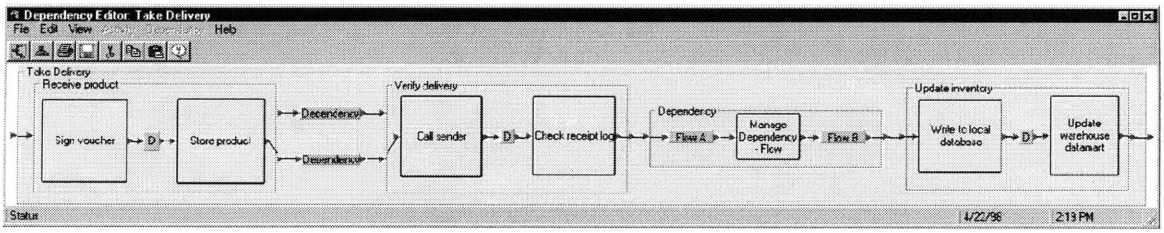

Figure 5-11: Replacing a Dependency with its Managing Activity

Note that the Dependency Editor does not require complete process specifications. Partial specifications are allowed which gives users the option to complete

their process descriptions as the theoretical details become available.

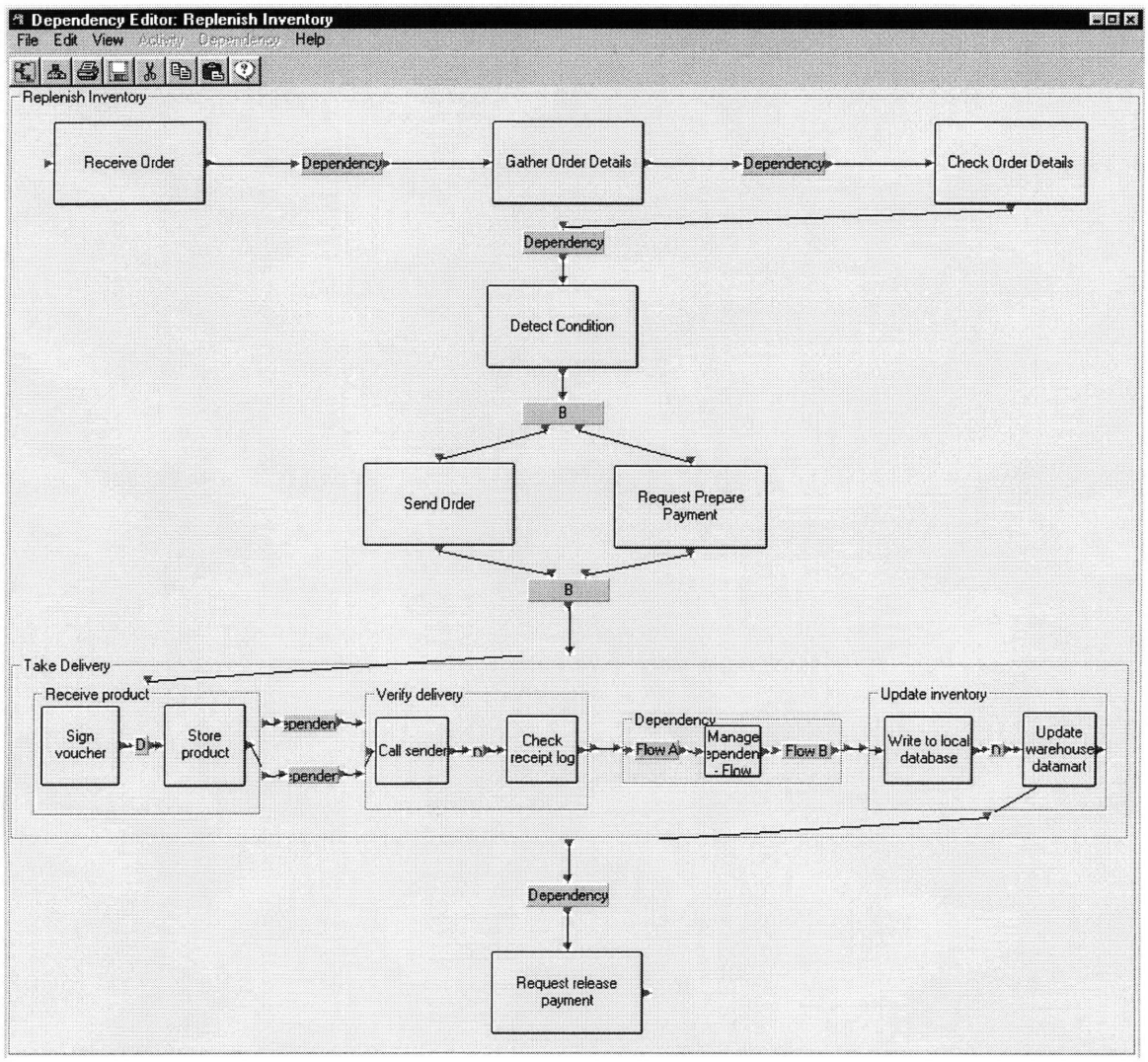

Figure 5-12: Complete View of the *Replenish Inventory* Process

Chapter 6

Future Work

6.1 Theoretical Constructs

Replacing a dependency with its managing coordination process is a common operation in the Dependency Editor. The exact semantics of this operation can be realized in two different ways. Figure 6-1 shows a simple flow dependency existing between two processes in the context of some high-level process.

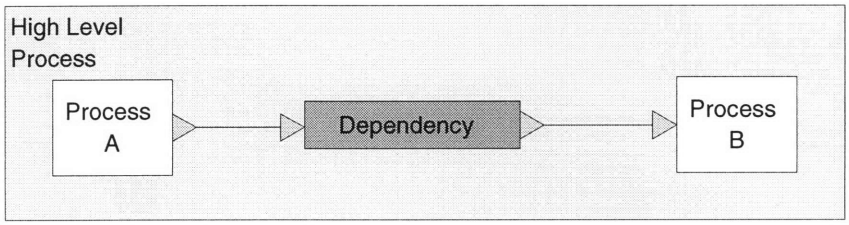

Figure 6-1: A Flow Dependency

Consider what happens when this dependency is replaced by its managing process. Assume that the managing process has already been defined and a unique mapping exists between the ports of the dependency and the ports of the managing activity. Figure 6-2 illustrates this process in the current model of the Dependency Editor.

The dependency remains visible but the managing process and synthetic flow dependencies are visible within its frame. The ports of the flow dependencies are

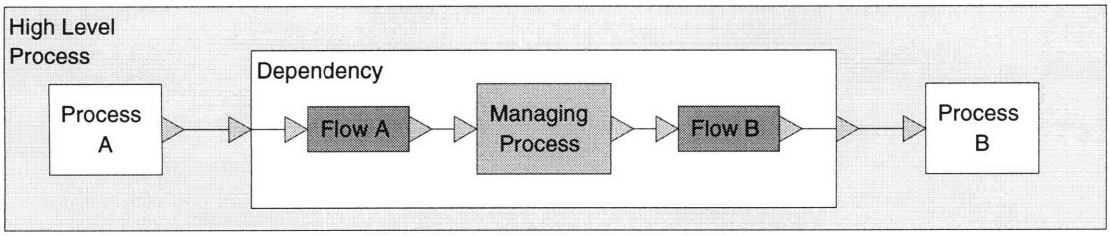

Figure 6-2: Replacing a Dependency with its Managing Activity

connected to the ports of the main dependency. An alternate representation is shown in Figure 6-3.

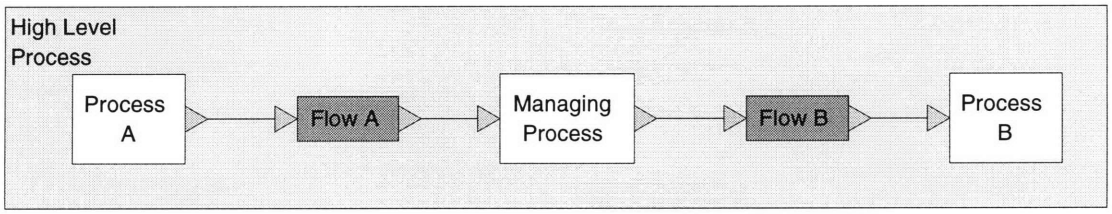

Figure 6-3: Replacing a Dependency: Alternate View

The dependency is replaced with two simpler flows, whose ports are connected to the ports of the original dependency. A noticeable difference is that the newly created synthetic dependencies are now connected to the connection nodes of the original dependency rather than to the dependency itself.

An immediate argument against the current model is that it conflicts visually with the dimension of dependency decomposition. Some cues need to be provided to the user to differentiate between the managing process and decomposition frames of a dependency.

6.2 Advanced Functions

Currently, the Dependency Editor does not support decomposable ports. This is not a flaw in the Handbook semantics – ports being first class objects in the Process Handbook can have a decomposition just like any other entity. The weakness stems from the difficulty of displaying this decomposition information. In the current worlds-within-worlds model, activity decompositions are rendered as boxes within the frame of the parent object. Ports are rendered as smaller structures on the sides of the parent entity. Decomposable ports do not fit into model, since port renditions are much to small to support variable depth decompositions. Additional work is required to develop a click-and-replace metaphor implementation of port decompositions.

Some work is needed in the Editor to support advanced port compatibility tests. Currently, the only compatibility checks performed relate to the port type; i.e. in the appropriate context, only the semantically permitted connections are allowed. For example, vertical connectors may only connect ports of the same type (e.g. Producer) and the type of the ports' parents must also be the same (e.g. ports of activities, one in the immediate decomposition of the other). Advanced compatibility checks would consider the type of resource flowing through the port, and connections would be allowed only if the resource types of two ports were similar or compatible.

Another very useful function would be the provision of automatic layouts. The function would be map an existing screen layout to one which minimizes screen clutter by some aesthetic measure. Efficient automatic layout algorithms have been presented in literature. The implementation of one – a significant project in itself – would be a great boon for increasing the usability of the Process Handbook.

Chapter 7

Conclusion

The Dependency Editor is a robust and flexible tool for process representation, analysis, and engineering. The Editor emphasizes dependency relations between various processes and the coordination processes which manage these dependencies. Users are provided with the functionality of laying out a process description on a plane, using as basic building blocks either existing processes in the Process Handbook or user-defined activities.

The Dependency Editor exposes users to the powerful representational paradigms of abstraction and hierarchy. Specifically, abstraction is employed by giving the user the ability to view only as much detail of a process as needed. Sub-activities in the decomposition of a process are hidden from sight unless specifically requested. A dependency – which is essentially a high-level representation of its managing process – can be replaced with the managing process upon request. This defines an additional abstraction mechanism whereby the underlying complexity of the managing coordination process and simpler dependencies is hidden from view unless necessary. This abstraction does not result in data loss – underlying detail in a process is hidden only until it is requested. If necessary, a user can view a process in the finest possible detail that has been defined in the Process Handbook semantics.

The concept of hierarchy is employed by giving users the functionality to engineer process descriptions in a bottom-up manner. Once a process has been defined, it can essentially be considered as a black box by the rest of the world – its underlying

implementation need not be exposed. This process can be carried out iteratively by building successively complex processes from simpler ones. The paradigms of abstraction and hierarchy together make the Dependency Editor an exceedingly useful tool for specifying process descriptions in the Process Handbook semantics.

The Dependency Editor is flexible in the sense that it allows partially defined process descriptions. Users are not bound to completely define a process by specifying managing processes for all dependencies or connecting all requisite ports of a dependency or activity. All in all, the Dependency Editor contributes towards making the Process Handbook a useful, extensible, and robust tool for analyzing processes.

Bibliography

[1] *DCOM: A Business Overview.* Microsoft Corporation, Technical Report. Available on the Internet, http://www.microsoft.com/ntserver/guide/dcom.asp, 1998.

[2] C. Dellarocas. *A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating Software Components.* PhD thesis, MIT, Department of Electrical Engineering and Computer Science, 1994.

[3] Y. Elley. *A Flexible Process Editor for the Process Handbook.* Master's Thesis, Department of Electrical Engineering and Computer Science, MIT, 1996.

[4] U. Farooq. *An Object Server for the Process Handbook.* Master's Thesis, Department of Electrical Engineering and Computer Science, MIT, 1997.

[5] J. Lee and G. Yost. *The PIF Process Interchange Format and Framework Version 1.1.* Workshop on Ontological Engineering, ECAI: Budapest, Hungary.

[6] T. Malone. *How Will You Manage in the 21st Century? From Command and Control to Cultivate and Coordinate.* Center for Coordination Science, MIT, 1996.

[7] T. Malone and K. Crowston. *What is Coordination Theory and How Can It Help Design Cooperative Work Systems.* Working Paper 112, Center for Coordination Science, MIT, 1990.

[8] T. Malone and K. Crowston. *Toward an Interdisciplinary Theory of Coordination.* Technical Report 120, Center for Coordination Science, MIT, 1991.

[9] T. Malone, K. Crowston, B. Pentland, J. Quimby, and A. Bernstein. *Tools for Inventing Organizations: Towards a Handbook of Organizational Processes.* Working Paper 225, Center for Coordination Science, MIT, 1997.

[10] S. Polyak and J. Lee. *Applying the Process Interchange Format (PIF) to a Supply Chain Interoperability Scenario.* Available on the Internet, 1998.