A Flexible Process Editor for the Process Handbook

by

Yassir Elley

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 7, 1996

Copyright 1996 M.I.T.
All rights reserved.
A Flexible Process Editor for the Process Handbook

by

Yassir Elley

Submitted to the
Department of Electrical Engineering and Computer Science

May 7, 1996

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 at populating an on-line "process handbook" of business process descriptions. The Process Handbook supports several editors, each of which concentrates on a different aspect of the methodology. This thesis describes the design and implementation of the Dependency Viewer, a flexible process editor which places emphasis on the dependency relationships that exist between activities and the coordination processes that manage these relations. There are several main features supported by the Dependency Viewer. The system provides flexibility by enabling users to arbitrarily lay out a process on a two-dimensional plane. An activity can be replaced with its decomposition, while maintaining finest grain dependency rendering. A dependency can be replaced by its managing process, revealing the particular surface structure of the process being studied. Combining these two features together, a composite successive replacement feature is enabled which allows users to zoom in on a particular area of interest while preserving the peripheral context. Finally, the system provides support for specifying and rendering an extendable list of dependency types, as well as decomposable dependencies, yielding a richer abstraction language with which to specify process descriptions.

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

I would like to thank many people who have helped me throughout the writing of my thesis. Firstly, I would like to thank my thesis advisor, Professor Thomas Malone, who has helped me greatly and supported my research with much enthusiasm. Thanks to John Quimby for helping me throughout the whole process from beginning to end and contributing heavily to the project, and thanks to George Wyner for his interesting theoretical insights and ideas. Thank you to all the people at the MIT Center for Coordination Science who have helped me along the way and have made it such a pleasant place to work. My heartfelt thanks to my parents, who have taught me the value of hard work, integrity, and humility, and who have made great sacrifices to provide me with the best education possible. I could not have made it without their support and encouragement. My warm thanks to my baby sister, my brother, and my sister-in-law, for their affection and guidance. Finally, my special thanks to my fiancee who has helped me through this challenging year and always supported me unconditionally with warmth and kindness. Most of all, I would like to express my gratitude to the Almighty for the countless blessings He has bestowed upon me over the years.
Table of Contents

1 Introduction .............................................................................................................. 6
2 Process Modeling Tools .......................................................................................... 8
   2.1 Introduction ........................................................................................................ 8
   2.2 Flowcharts ........................................................................................................ 9
   2.3 Petri Nets .......................................................................................................... 11
   2.4 Dataflow Diagrams .......................................................................................... 14
   2.5 SADT Diagrams ............................................................................................... 17
   2.6 Other Interesting Tools .................................................................................... 19
       2.6.1 Improvise ................................................................................................. 20
       2.6.2 Envision .................................................................................................... 20
       2.6.3 STATEMATE ........................................................................................... 21
       2.6.4 SYNTHESIS ............................................................................................ 21
3 The Process Handbook ............................................................................................ 24
   3.1 Coordination Theory ........................................................................................ 24
   3.2 Project Overview and Goals ............................................................................ 25
   3.3 Process Handbook Methodology ..................................................................... 25
       3.3.1 Abstraction and Inheritance ..................................................................... 26
       3.3.2 Managing Dependencies .......................................................................... 30
4 System Overview .................................................................................................... 32
   4.1 Desirable Properties ........................................................................................ 32
       4.1.1 Graphical User Interfaces ......................................................................... 32
       4.1.2 Multiple Editors ........................................................................................ 32
       4.1.3 Robust Database Schema ......................................................................... 33
       4.1.4 Compatibility ............................................................................................. 33
   4.2 Implementations ................................................................................................ 34
       4.2.1 Kappa PC Implementation ........................................................................ 34
       4.2.2 Lotus Notes Implementation ...................................................................... 34
       4.2.3 Current Implementation ............................................................................ 35
   4.3 Dependency Viewer ........................................................................................... 35
       4.3.1 Activities .................................................................................................... 36
       4.3.2 Dependencies and Resources ..................................................................... 36
       4.3.3 Ports ........................................................................................................... 37
5 Replacing an Activity with its Decomposition ....................................................... 40
   5.1 Discussion .......................................................................................................... 40
       5.1.1 Abstraction ................................................................................................. 40
       5.1.2 Decomposition Depth ................................................................................. 41
       5.1.3 Decomposition Focus ............................................................................... 41
       5.1.4 Decomposition Consistency ...................................................................... 41
   5.2 Implementation .................................................................................................. 42
       5.2.1 Ports and Dependencies ............................................................................. 42
       5.2.2 Ports and Connectors ................................................................................ 44
       5.2.3 Bottom-Up Specification and System Generation of Connectors ............. 47
       5.2.4 Context and Multiple Dependency Viewers .............................................. 49
Chapter 1

Introduction

The Process Handbook project at the Center for Coordination Science aims at populating an on-line "process handbook," using field data that shows how different organizations perform similar activities, as well as the relative advantages and disadvantages of each approach (Malone, et al., 1993). The main goals of this handbook are to help theoreticians imagine new organizational structures and to help consultants redesign existing organizations. Using the handbook's methodology, differences and similarities between related processes can be represented explicitly, allowing users to choose from among a variety of alternative ways in which a particular process could be performed, along with supplementary information indicating the potential advantages and drawbacks of each alternative. Taking field data from a broad range of industries and organizations, the handbook can be a powerful tool, in that it allows users to consider alternative ways of performing a process that they may never have considered before.

The handbook project draws heavily from coordination theory in the sense that it views coordination processes as ways of managing dependencies between activities. By building and maintaining a generic list of various coordination processes that manage particular dependency types, the handbook would serve as a repository, mapping each dependency type to a collection of coordination processes that could manage it. Thus, when a new process is being analyzed, composed of several activities and various dependency relationships between the activities, the handbook would generate a set of possible coordination mechanisms for each dependency. These alternatives suggested by the system may be non-obvious, allowing the user to analyze the situation from a different perspective. Furthermore, creating this set of possible coordination processes would not be a function of the particular process that was being analyzed, but rather would only be related to the specific dependencies that are present in that process.
Additionally, the handbook borrows ideas about inheritance from computer science, establishing specialization and decomposition relationships between related activities. The present implementation of the Process Handbook does have substantially developed viewers that emphasize the inheritance relationships between activities (including both specialization and decomposition relations) and effectively manage these complex relations. However, the present implementation does not have adequate support for the other major component of the representational methodology (i.e. that of managing dependencies) and it is to this end that this thesis project is aimed.

The work performed for this thesis was to design and implement a robust mechanism for viewing, analyzing, and editing dependency structures and the coordination mechanisms that manage them within the Process Handbook. As opposed to the existing viewers, this coordination editor emphasizes the dependency relations between activities rather than the inheritance relations.

This paper describes the thesis project that was undertaken. In Chapter 2, background information is provided about the field of process modeling and four prominent process modeling approaches are examined. Chapter 3 gives a summary of the Process Handbook project and the methodology that it uses to represent processes. Chapter 4 provides a general overview of the system that was implemented. The next several chapters are devoted to discussing the main features of the Dependency Viewer. Chapter 5 discusses the capability and implications of allowing users to replace an activity with its decomposition, while maintaining finest grain dependency rendering. In a similar manner, Chapter 6 examines the feature of replacing a dependency with its managing process. Chapter 7 describes the support provided by the system to allow a variety of dependency types, as well as decomposable dependencies, to be specified and rendered. Finally, Chapter 8 discusses future work that can be undertaken that would enhance the functionality of the Dependency Viewer, and Chapter 9 draws the paper to an end with some concluding remarks.
Chapter 2

Process Modeling Tools

2.1 Introduction

This section places the project in context by introducing the field of process modeling and discussing related work in this domain. Specifically, four prominent modeling approaches will be examined in turn, namely flowcharts, petri nets, dataflow diagrams, and the SADT methodology. Additionally, examples of software tools that support each of these approaches will be given, along with some other interesting examples that do not fit neatly into any of these four categories.

A process model is a representation of a process that focuses on certain important aspects while downplaying less critical ones. Process models can be used to facilitate understanding and communication (Curtis, et al., 1992). The model can enable different groups of people to precisely discuss a particular process by using a common, formalized representational format. By examining and analyzing the model, people can gain new insights into the process, leading to process improvement. Clearly, making changes and experimenting with the model is much safer, more convenient, and less expensive than manipulating the actual process.

Process models can be created, edited, and manipulated using modeling tools. Before examining four popular modeling approaches, it is useful to mention some properties that good modeling tools should have (Yourdon, 1989). A modeling tool should use graphical visual representations to model the components of the process, as well as the interfaces and interactions between these components. Supporting text is also essential to provide the actual details of the process. In order to deal with complex processes, the tool should allow the process to be viewed at different levels of detail, in a top-down partitioned fashion. It should have minimal redundancy so that the model can be easily updated. Finally, it
should create a model which is transparent and easily understandable.

2.2 Flowcharts

There are two commonly used types of flowcharts (system flowcharts and logic flowcharts) which can be used in conjunction with each other to model a process or system (Jeffery and Lawrence, 1984). System flowcharts provide a high-level representation of the process, breaking up the larger process into smaller subprocesses. The logic of a particular subprocess is then modeled using logic flowcharts, which were specifically developed to model sequential, procedural logic. Figure 2.1 shows the relationship between the two types of flowcharts. The figure shows part of an inventory system being modeled using a system flowchart with a logic flowchart describing the detailed logic of the “print order list” process.

System flowcharts focus on the processes that make up the larger process being studied, as well as their inputs, outputs and interconnections. Usually, rectangular boxes are used to represent the individual subprocesses, while various icon-like symbols are used to represent the physical resources (which could be inputs or outputs). The flow of these resources between subprocesses is described by the interconnecting flowlines. Using these symbols, a system flowchart attempts to model the flow of resources between the processes that make up the larger process.

Logic flowcharts are used to describe the details of what each high level process in the system flowchart actually does (Yourdon, 1989). In a commonly used approach, there are only three components to the notation. The parallelograms represent executable steps that need to be performed, while the diamond-shaped boxes represent decisions that need to be made. The flow of control is modeled with the directed arrows that connect the boxes. In general, in logic flowcharts, only one arrow can leave a parallelogram and only two arrows can leave a decision.
**Figure 2.1:** Relationship between the system flowchart and the program (logic) flowchart. Taken from (Jeffery and Lawrence, 1984).
There are several commercial tools that provide support for the diagramming of flowcharts. Some popular tools include Visio Corporation's Visio, MicroGrafx's ABC Flowcharter, and Scitor's Process Charter. These tools enable the user to select from a palette of shapes to quickly and effectively create flowcharts. ¹

2.3 Petri Nets

Petri nets are a second example of a modeling approach used to represent processes. A Petri net is a bipartite directed multigraph which is composed of two different kinds of nodes, named places (represented by circles) and transitions (represented by bars) (Peterson, 1981). The two kinds of nodes are connected by directed arcs, with the bipartite constraint that an arc can only connect a place and a transition. That is, two places can never be connected together, nor can two transitions. Tokens are primitive objects in Petri nets which reside in places and which control the execution of the net. A net executes by firing transitions, which has the effect of removing the tokens from the input places of the fired transition, and depositing a new token to each of its output places. A transition can only fire if it is enabled, which means that it has at least as many tokens in each of its input places as there are arcs coming to it from each of its input places. Figure 2.2 gives an example of a simple Petri net and illustrates the firing rules. Transitions t1, t3, and t4 are enabled since each of them has an adequate number of tokens in their respective input places. Transition t2 is not enabled, since two of its input places (p2 and p3) contain no tokens.

¹ For more information about these tools, refer to (Jantz, 1993), (Powell, 1995), and (Tyo, 1995) respectively.
Figure 2.2: A marked Petri net to illustrate the firing rules. Transitions t1, t3, and t4 are enabled. Taken from (Peterson, 1981).

Petri nets can be interpreted as representing a process where a set of conditions determine which events can occur, and a new set of conditions is produced after an event occurs (Maiocchi, 1983). In the process modeling context, the places of a Petri net can be thought of as resources and its transitions can be thought of as activities. An activity can only execute when its inputs are available, and upon executing, an activity consumes its inputs and produces outputs. Figure 2.3 gives a simple example of a Petri net used to model a process in a medical analysis laboratory.
The acceptance activity can only take place when there is an applicant present and when the receptionist is available. After execution, the acceptance operation produces a test request form, which is capable of firing the drawing operation, if a doctor and patient are also present. The drawing activity outputs a sample and an updated test request form. Only after the sample has been analyzed and has outputted a result can the reporting activity take place, leading to the production of the final report.

Petri nets are able to effectively model several important characteristics of a process. Con-
currency and sequentiality in time are well modeled, as are causal dependency and independence. Production, occupation, and consumption of resources are also inherently modeled. Petri nets are particularly well suited, but not limited, to modeling processes where activities occur asynchronously, concurrently, and independently. Possible areas that Petri nets could help model include resource allocation, operating systems, queueing networks, traffic control, and distributed systems. Finally, it is important to note that Petri nets can also be used to hierarchically model a system (Peterson, 1977). Both abstraction and refinement are supported, with an entire net being replaced by a place or transition in the former case, and a single place or transition being replaced by subnets in the latter.

There are many software tools that facilitate the creation and analysis of petri net diagrams. Commercial products that use petri nets to represent business processes include Process Weaver by Cap Gemini Innovation and PROMATIS's INCOME. The development of the Macrotec toolset at CRIM is an example of a research project in this field. 2

2.4 Dataflow Diagrams

Dataflow diagrams concentrate on the flow of data, as the name suggests, and also on the transformation of inputs into outputs (Yourdon, 1989). As opposed to flowcharts, dataflow diagrams do not explicitly represent the sequential logic of a process. The diagrams are made up of processes, flows, data stores, and terminators. Processes, shown by circles, represent the various functional units of the overall process that transform inputs into outputs. Directed named arrows are used to show flows, which connect the processes, and which represent the movement of data (or, alternatively, resources) around the system. Data stores are shown by two parallel lines and represent static data that needs to be remembered. In reality, stores are often implemented as databases or files. Finally, terminators, shown as rectangles, portray the data sources or data end users that are external to the context of the process being studied. The diagramming vocabulary outlined above is based on a widely used approach, although variations do exist.

2. For more information about these tools, refer to (Fernstrom, 1993), (PROMATIS Informatik, 1996), and (Keller, et al., 1993) respectively.
Data flow diagrams are organized in a hierarchy of levels, where a portion of a diagram can be exploded on a lower level diagram, showing progressively more detail. The topmost diagram, called the context diagram, establishes the external interface between the process and those entities outside the context of the process. The next diagram, called “Figure 0”, portrays the highest level view of the process and identifies the most prominent processes that make up the larger system. All significant flows, data stores, and terminators are also added to this diagram and the processes are numbered. Each of these processes on “Figure 0” is then individually decomposed to show a detailed view of its component processes, along with their associated flows, stores, and terminators. This decomposition procedure is continued until a lowest level set of diagrams is created with each process on these diagrams representing a single task. Figure 2.4 shows a set of leveled dataflow diagrams and also illustrates the numbering scheme which is used to keep track of which decomposition refers to which parent process.
**Figure 2.4:** Leveled dataflow diagrams. Taken from (Yourdon, 1989).

Dataflow diagrams are a main tool taught in courses on systems analysis. As such, several commercial software products exist which support the creation and editing of dataflow diagrams. These tools include Digital Insight’s Robochart, HavenTree’s EasyFlow, and Westmount Technology’s Westmount I-CASE Yourdon.  

---

3. For more information about these tools, refer to (Digital Insight, 1996), (Patz, 1995), and (Westmount Technology, 1996) respectively.
2.5 SADT Diagrams

The SADT (Structured Analysis and Design Technique) diagram (a subset of which is widely used by the government and is called IDEF0) is really a variation of the classic dataflow diagram. The SADT diagram builds on the dataflow diagram, introducing a number of additional features (Marca and McGowan, 1988).

SADT diagrams consist of boxes, which represent activities, and arrows, which specify the interfaces and interconnections of the activities with other activities. As a further distinguishing feature from other modeling approaches, each side of an SADT box, shown in Figure 2.5, has a certain, specific meaning.

![Figure 2.5: An SADT box](image)

Inputs always enter from the left side of the box and outputs always leave from the right side. The top side is specially reserved for controls and the bottom side is reserved for mechanisms. An SADT box can be interpreted in the following manner: inputs are transformed into outputs, provided the supporting mechanism (which specifies how activities are actually performed) is available, and only under the conditions constrained by the controls. Arrows represent collections of significant things and are used to connect boxes
together. Arrows can split apart and join together to portray how parts of the collection interact with the system. As with dataflow diagrams, SADT diagrams utilize a decomposition hierarchy, using a similar numbering scheme to keep track of pairs of parent-child diagrams.

One major difference from dataflow diagrams is that SADT diagrams distinguish between the inputs that are transformed by a process and the controls that govern the transformation. An example is given in Figure 2.6, which shows an Assemble activity that transforms a set of inputs into a chair (Marca and McGowan, 1988). Without the blueprint control, the activity could be accomplished in any number of imaginable ways. However, the explicit presence of the blueprint control places a constraint on the activity, dictating that the inputs must be transformed into the chair according to the blueprint.

![Blueprint](image)

**Figure 2.6:** Separating Controls from Inputs. Taken from (Marca and McGowan, 1988).

SADT diagrams also place emphasis on the mechanisms which are used to accomplish the activity. Continuing with the previous example, if a glue mechanism arrow was added to the bottom side of the Assemble activity box, this would determine how the seat would physically be assembled. Replacing the glue mechanism with a screwdriver mechanism
would significantly alter how this process would be executed. Mechanisms can also be employed to specify the actors involved in the process. Two or more mechanism arrows connected to a box can represent a coordinated activity, where an activity is accomplished by several people working together. Mechanism arrows can also be used to implicitly represent sequencing. If the output of activity X serves as a mechanism for activity Y, then clearly activity X has to complete execution before activity Y can begin, since activity Y needs all its mechanisms in place before it can even start execution.

Another fundamental feature of SADT diagrams is that they allow dataflow feedback to be differentiated from control feedback. Dataflow feedback occurs when the output of one activity serves as the input of another activity, whose output affects the original activity in some way. Dataflow feedback is often used to allow for recycling and iteration. Control feedback happens when the output of one function controls a second function, whose output controls the first function. Control feedbacks can be used to optimize a process by updating available control information and using negative feedback. Since both activities constrain each other, it is possible that the system will fail if the feedback is not maintained. Clearly, control feedbacks need to be monitored more closely than dataflow feedbacks.

Since the SADT/IDEF0 methodology is so widely used, there are several commercial software tools that have been designed to assist users in creating, modifying, and storing process models using this methodology. These tools include ProcessWorks! by Wizdom Systems, BPWin by Logic Works, and METIS GEM by METIS. ⁴

2.6 Other Interesting Tools

This section discusses a few additional software tools that aid users in business process modeling, but that do not fit neatly in the modeling approaches discussed so far. A brief survey of three software CASE tools is presented, along with a summary of recent doc-

---

⁴ For more information about these tools, refer to (Wizdom Systems, 1996), (Mace, 1993), and (Hoyte, 1993) respectively.
toral work undertaken at the Center for Coordination Science at MIT.

2.6.1 Improvise

Improvise, developed by AT&T, is a multimedia system for modeling, displaying and documenting business processes (Barghouti, 1996). The primary goal of Improvise is to create a system that is as easy and simple to use as commercial workflow management systems, but yet at the same time as powerful as process-centered software engineering environments. To realize this goal, Improvise enables users to graphically describe a process flow diagram. Additionally, users are allowed to associate multimedia information and executable attachments to any node (activity) or edge (arrow) of the diagram. At the user’s option, the layout of the diagram can be computed automatically or can be flexibly specified by the user. The set of shapes used in the graphical language is customizable and is particularly suited for business processes in which there is information flow. By default, these shapes include node types to represent tasks (activities), systems, sources/destinations of data, documents, decision points, and connectors between different processes. The objects supported by the system are nodes and edges, each of which has an associated set of attributes whose values can be assigned interactively. Some of these attribute values are used to automatically compute overall process characteristics, such as time and cost. Processes can also be replaced by their subprocesses. Finally, Improvise follows the open system philosophy, allowing it to be easily connected to other tools, such as process simulation and performance analysis tools.

2.6.2 Envision

Future Tech’s Envision software tool allows processes to be modeled as collections of objects which can be connected together by connectors (Mann and Stucki, 1994). Every item on a diagram is either an object or a connector. The diagramming metaphor is that of a zoomable sheet of paper, with each sheet having a palette of objects from which to draw. There are pre-defined palettes (such as for entity-relationship diagrams and dataflow diagrams), as well as user-definable palettes. Each palette is associated with semantic rules for connecting the items on the palette. The objects themselves are unique entities in the
underlying database, and thus, the same object may appear in different diagrams using different diagramming vocabularies. Thus, Envision enables the creation of an enterprise model with multiple views of the same data.

2.6.3 STATEMATE

STATEMATE also provides support for multiple representational paradigms, motivated by the belief that different models are able to effectively highlight different features of a process (Curtis, et al., 1992). STATEMATE-based modeling provides formal visual languages with which to create the models. The models can then be displayed using a multiply integrated perspective and can be put through automated analysis and simulation features. STATEMATE is designed to facilitate human understanding and communication about the process, support process improvement and support process management. The software allows processes to be represented by three major language paradigms. It supports state transition diagrams with events and triggers in order to represent the behavioral perspective. For the functional perspective, it supports dataflow diagrams and SADT diagrams, and entity-relationship diagrams. Finally, it provides support for modified structure charts, to allow the organizational perspective to be modeled. The STATEMATE modeling approach has been effectively applied to model the actual processes that are used to support the operational software used in the F-14A and F-16A/B aircraft.

2.6.4 SYNTHESIS

The work that is most closely related to this present thesis is the doctoral work performed by Chris Dellarocas, a former graduate student at CCS. Dellarocas applied many of the ideas of the Process Handbook and coordination theory, viewing software applications as a collection of software activities and dependencies, which eventually need to be managed by coordination processes (Dellarocas, 1996). He has created an architectural description language for describing software applications called SYNOPSIS, which uses the same ideas of separating activities from dependencies and the notion of entity specialization that are used in the Process Handbook. The application development tool he developed, called SYNTHESIS, keeps a repository of increasingly specialized dependency types and coor-
dination processes that can be used to manage these dependencies in the specialized domain of software component integration. SYNTHESIS facilitates the generation of executable applications by semi-automatic transformations of their SYNOPSIS descriptions. Specifically, activities are successively replaced by more specialized activities and dependencies are managed by coordination processes, until the process description is precise enough that the system can generate code to create an executable system.
Chapter 3

The Process Handbook

3.1 Coordination Theory

Before examining the Process Handbook and its methodology for representing processes, a brief discussion of coordination theory is helpful, since the Handbook draws heavily on major parts of coordination theory.

Coordination theory focuses on the interdisciplinary study of coordination, using and extending ideas about coordination from areas such as computer science, organization theory, operations research, economics, linguistics, and philosophy (Malone and Crowston, 1991). Broadly speaking, coordination can be defined as the act of working together. In a narrower context, focusing specifically on those characteristics of a system unique to coordination, coordination is defined as the act of managing interdependencies between activities.

To analyze a situation from the perspective of coordination theory, it is useful to identify four elements: goals, activities, actors, and dependencies (Malone and Crowston, 1991). Goals can be thought of as the desired end results or other evaluation criteria. In order to achieve the goals, however, a series of activities need to be performed by an individual actor or multiple actors. For coordination to be needed, it is implicit that these activities are not independent, but rather are related to each other in some way. The whole field of coordination theory is largely centered around the concept of viewing coordination processes as being ways of managing the dependencies between these activities, with the aim of achieving a set of goals.

In a typical application of coordination analysis, these four elements can be identified in the order given above. Firstly, a set of goals is identified. The goals then need to be
decomposed into various activities which need to be performed in order to reach the goals. These tasks are then mapped to actors who actually execute the tasks. Finally, and most importantly from the point of view of coordination theory, the coordination processes for managing the dependencies between these activities need to be identified. It is in this final step that the actual “coordination” takes place and it is this final step in which coordination theory is most interested. In that regard, research in this field includes characterizing different kinds of dependencies and identifying the coordination processes that can be used to manage them.

3.2 Project Overview and Goals

The Process Handbook project at the Center for Coordination Science aims at populating an on-line “process handbook,” using field data that shows how different organizations perform similar activities, as well as the relative advantages and disadvantages of each approach. The main goals of this handbook are to help theoreticians imagine new organizational structures and to help consultants, managers, or workers redesign existing organizations. Using the handbook’s methodology, differences and similarities between related processes can be represented explicitly, allowing users to choose from among a variety of alternative ways in which a particular process could be performed, along with supplementary information indicating the potential advantages and drawbacks of each alternative. Taking field data from a broad range of industries and organizations, the handbook allows users to consider alternative ways of performing a process that they may never have thought of before. For example, a user may be interested in systematically examining all the possible alternatives of how consulting services could be provided. Using the handbook, the user would be able to find not only the standard practices used by consulting firms of directly selling their services to selected clients, but would also find techniques using mail advertising, or perhaps even more radical alternatives such as retail storefront sales.

3.3 Process Handbook Methodology

The key intellectual challenge of the Process Handbook project is to develop a methodology capable of robustly representing processes (Malone, et al., 1993). By explicitly repre-
senting the similarities and differences between related processes, the language enables the user to leverage off of existing processes. New related processes would not have to be specified “from scratch” but could simply be represented by noting how they differ from similar processes that have already been modeled. For example, if the model for a generic “order-entry process” already exists in the handbook, then a specific order-entry process could be analyzed and represented by the features which distinguish it from the generic process. Using this framework, a list of alternative processes that could be used in a particular scenario could be easily displayed. In order to realize this framework, the methodology uses ideas from computer science about inheritance, as well as concepts from coordination theory about managing dependencies.

3.3.1 Abstraction and Inheritance

One novel feature of the handbook is that it represents processes at different levels of abstraction. Processes inhabit places on a specialization hierarchy, which arranges the processes from the more generic level to the more specific level. As with traditional object-oriented hierarchies, a specific “child” process automatically inherits the properties of its more general “parent” process. These inherited properties can then be altered or removed from the child process, as appropriate, and new properties unique to the child process can be added. Additionally, a process can be decomposed into subactivities, showing a more refined view of the functions that are carried out by the process. Figure 3.1 illustrates the interplay between specialization and decomposition.
Figure 3.1: Relationship between specializations and decompositions. Adapted from Malone, et al. (1993).

B and C are alternative specializations of A. A is replaced by B in the specialization A must precede B.

A is replaced by B in the specialization A is inherited in the specialization.
The generic “Sell Product” process is decomposed into its five subactivities, including “Identify prospects” and “Inform prospects about product.” As Figure 3.1 indicates, the “Sell Product” activity is also specialized into more specific activities like “Direct mail sales” and “Retail storefront sales,” which can be interpreted as alternative ways of performing the parent activity. Each of the specializations inherit the decompositions of their parent activity. In both cases, however, the inherited decompositions have been modified to reflect differences between the specialized activities and their parent activity. For example, in the decomposition of “Direct mail sales,” although “Obtain order” and “Deliver product” are inherited without alteration, the activity “Obtain mailing lists” has replaced “Identify prospects.” Although the diagram does not show the relation, “Obtain mailing lists” is a specialization of “Identify prospects.” In fact, if an inherited activity is replaced in a decomposition, it must always be replaced by one of its specializations. Similarly, since retail storefront sales do not require orders to be obtained nor prospects to be identified, these activities are omitted from the decomposition.

A few words about inheritance seem in order at this point. The “properties” of a parent activity that are passed on to their children include the parent’s textual attributes (such as “actors involved in activity”, “goals of activity”, etc.), and also include more complex properties such as its decomposition, as well as the dependency structure between the subactivities of its decomposition. As Figure 3.1 indicates, modifications will usually be made to some aspect of the inherited decomposition. If absolutely no alterations are made, then it would not usually make sense to have created a new specialization, since it would be indistinguishable from its parent. The concept of context is extremely important when talking about modifications to a decomposition, since the same modification may have different effects depending on the context that is specified. As we would expect, alterations to the child will not be reflected in the parent, while modifications to the parent WILL show up in the child. For example, in the decomposition of “Direct Mail Sales”, “Obtain Order” is one of the activities that has been inherited in the specialization. If the “Obtain Order” activity is altered in the context of “Sell Product,” then it will also change in the same way in “Direct Mail Sales”, as well as in all other specializations of “Sell Product” where the activity is inherited. If however, it is changed within the context of “Direct Mail
Sales” (for example, by further decomposing it, or even by simply renaming it), then only “Direct Mail Sales” would show the change.

In order to make interesting comparisons, groups of alternative specializations can be combined into “bundles” of related alternatives, which are classified by the dimension of the situation they are describing. For example, “Sell product” may have a bundle of specializations having to do with how the product is sold (direct mail, retail storefront, telemarketing, etc.), a separate bundle relating to what product is sold (computers, financial services, apartments, etc.), and a third bundle concerned with to whom the product is sold (consumers, corporations, governments, etc.). Using a set of goals as the comparison criteria, the Process Handbook can display a trade-off matrix to compare alternatives within a bundle. Additionally, alternatives in a particular bundle automatically inherit alternatives from the other bundles, but do not inherit the other alternatives from their own bundle. For example, for someone selling chocolates, it might be useful for them to be automatically provided with the alternatives for direct mail sales and telemarketing, whereas presenting them with the alternative of selling computers would not make sense.

Using a combination of decomposition and specializations to represent processes has several advantages. Representing new processes can be greatly facilitated by this approach, since the user simply needs to introduce the new process as a specialization of a more general process. The user can take advantage of the fact that the new process inherits all of its parent’s properties by keeping the appropriate ones, and altering or creating new ones. Additionally, process descriptions can be more easily maintained since changes to high level processes are inherited by all their specializations, filtering down through the entire hierarchy. Furthermore, the concept of bundles and the trade-off matrix allows alternatives to be compared, enabling processes to be selected appropriately according to the situation at hand. Finally, by having a specialization hierarchy, a structure is imposed on the handbook allowing new processes to be organized and classified in an effective manner.
3.3.2 Managing Dependencies

The methodology also borrows heavily from coordination theory in the sense that it views coordination processes as ways of managing dependencies between activities. By building and maintaining a generic list of various coordination processes that manage particular dependency types, the handbook would serve as a repository, mapping each dependency type to a collection of coordination processes that could manage it.

The differentiation between dependencies and the activities that manage them is a novel feature of the methodology, significantly enhancing its representational abilities. It both enables a more concise representation of processes and also allows us to generate alternative possibilities for managing a particular dependency.

The conciseness of representation follows from the fact that all the coordination processes that manage dependencies in a particular process need not be listed separately in every process. Rather, a particular dependency in a process representation can be thought of as being managed by an instance of an appropriate coordinating process, which would have other instances managing the same dependency type in other process representations. Furthermore, a coordination process is composed of a collection of activities and dependencies. Clearly, it would be more concise to abbreviate this complex collection with a single dependency.

Thus, when a new process is being analyzed, composed of several activities and various dependency relationships between the activities, the handbook would generate a set of possible coordination mechanisms for each dependency. These alternatives suggested by the system may be non-obvious, allowing the user to analyze the situation from a different perspective. Furthermore, creating this set of possible coordination processes would not be a function of the particular process that was being analyzed, but rather would only be related to the specific dependencies that are present in that process.
Chapter 4

System Overview

4.1 Desirable Properties

Before discussing the actual Process Handbook modeling tool, it is helpful to highlight some of the key characteristics that the tool should possess in order to achieve the goals of the project and to enable the representational power of the Process Handbook methodology.

4.1.1 Graphical User Interfaces

The handbook should make extensive use of graphical user interfaces. The advantage of using visual process representations can not be overemphasized, as visual displays of information are able to concisely capture concepts, which would otherwise require large amounts of textual description. Of course, the handbook should also store supporting text, since these details would definitely be needed to adequately map out a process. Wherever possible, the tool should allow the user to graphically manipulate the elements of the process description (i.e. using drag/drop operations, dialog boxes, etc.). In order to encourage use, the tool should be user-friendly, using plain descriptive language in the command menus and clearly and cleanly defining the metaphors for manipulation in a way that is easily understandable and easily usable. This last point is especially important since the main goal of this tool is, after all, to facilitate users with process entry and process editing so that they may more fully understand the process being studied, hopefully gaining new insights with the help of the tool.

4.1.2 Multiple Editors

We would like the tool to have multiple editors to the underlying process descriptions. Espousing the philosophy that different views may be appropriate for different tasks, the handbook should allow for a variety of perspectives of the data to be displayed. At the
same time, the interface to each editor should be consistent with respect to the other editors, using the same visual codes and commands, so that usage is facilitated. Furthermore, it should also be possible to easily navigate and communicate between different editors. This would create a sense of unity and cohesiveness between the various parts of the tool. In addition to supporting different perspectives of the data, multiple views of the same perspective would also be useful (i.e. viewing several decomposition hierarchies simultaneously).

4.1.3 Robust Database Schema

There should be a rich and robust database schema to support and implement the main ideas of the methodology, namely abstraction, inheritance, and coordination mechanisms. The schema should be carefully designed so that relationships between the data are adequately and efficiently captured. Since we would like the tool to be more than just a simple viewer, it needs to be able to automatically manipulate the data appropriately behind the scenes. Inheritance support is a prime example where the system’s computational power can be exploited. As stated earlier, all specialized activities inherit the attributes, decompositions, and dependencies of their parent. Thus, a simple modification on a high level activity would have an enormous effect, as all of its children would inherit the modification. The tool should be able to support these and other equally complicated calculations. Furthermore, the calculations should be efficient and optimized to minimize processing time. Since a common platform for this tool might be on laptops, when users would be conducting field studies, gathering and entering process descriptions, it should be kept in mind that the processing power of a desktop computer may not always be readily available. Thus, efficient algorithms and coding techniques are a necessity. The system should also provide automated guidance for the user whenever possible, prompting the user with queries and initiating dialogs appropriately to facilitate usage.

4.1.4 Compatibility

To truly enable widespread usage of the tool, it is fundamentally important that the process descriptions encoded by the handbook’s methodology be compatible with other methodol-
ologies. The Process Interchange Format (PIF) is currently being developed by researchers to address this issue (Lee, et al., 1994). PIF serves as a common process ontology, from which processes can be translated to any other process format. Thus, if we wanted to load a process description into the handbook that had been created and edited using a representation different from the handbook’s representation, we would first pass the description through a PIF translator, which would convert it into the globally compatible PIF format. We would then translate the process from PIF to the handbook’s representation, allowing us to view, edit and manipulate the description using the handbook’s tools and features. In a similar manner, handbook entries could be extracted into PIF and translated into some other representation so that they could be manipulated with some other modeling tool.

4.2 Implementations

This section briefly outlines some of the earlier implementations of the handbook, before going on to examine the current implementation.

4.2.1 Kappa PC Implementation

The earliest prototype implementation of the Process Handbook was built by Chris Dellarocas, a former doctoral student at the Center for Coordination Science, using Intellicorp's Kappa PC knowledge-based application building tool. This application was able to adequately provide most of the desired functionality required by the methodology. However, the implementation was limited in its capability to handle a large number of processes, as well as large amounts of structured text.

4.2.2 Lotus Notes Implementation

The second implementation was developed by George Wyner, another CCS doctoral student, using the popular groupware software package, Lotus Notes. This version corrected the shortcomings of the first prototype, in that it was able to handle large numbers of semi-structured process descriptions, but only at the expense of introducing equally serious limitations. Namely, the application was unable to support the concept of process inheritance among activities, which is fundamental to the methodology.
4.2.3 Current Implementation

The current implementation of the Project Handbook has survived for a much longer time than the earlier prototypes, with the consequence that many people have been and continue to be involved in its development, including the author. The current application has been developed using Visual Basic as the main authoring environment, along with Microsoft Access database files as the repository for process descriptions. This implementation exhibits the desired characteristics and features outlined earlier. Namely, it has a seamless, unified graphical user interface with multiple viewers, and provides support for abstraction, inheritance, and dependency structures and manipulation. Furthermore, it is capable of handling large amounts of information, thus overcoming the major limitations of the earlier prototypes.

Before this thesis project, the system already had two substantially developed editors, which emphasized the inheritance relationships among activities and effectively managed these complex relations. Specifically, the Specialization Viewer and the Decomposition Viewer each focused on the specialization and decomposition relations and hierarchies, respectively, and also allowed the user to navigate between the two perspectives. However, this old implementation did not have adequate support for the second major component of the representational methodology (i.e. that of managing dependency mechanisms) and it is this limitation that this thesis project attempts to remedy by creating the Dependency Viewer, which is the main deliverable of this thesis project.

4.3 Dependency Viewer

The Dependency Viewer offers an alternate view of a process description, where the dependency relations between activities are given emphasis, rather than the inheritance relations. The Dependency Viewer provides a robust mechanism for viewing, analyzing, and manipulating dependencies and the coordination processes that manage them. Furthermore, as opposed to previous editors, which were hierarchical in appearance, the
Dependency Viewer, for the first time, enables the user to flexibly lay out the process description on an unconstrained two-dimensional plane, yielding a more traditional and familiar process map display.

Before discussing and analyzing the main features of the Dependency Viewer, it is important to examine the primitive structures used by this viewer to specify process descriptions. The major primitives can be classified as activities, ports, dependencies, and resources.

4.3.1 Activities
The Dependency Viewer provides the same notion of abstraction supported by the other viewers. Specifically, activities can be decomposed into finer subactivities to show a more refined level of functional detail, which can be further decomposed to show even more detail, and so on ad infinitum. This is all we would have to say if we were simply specifying the major functional units that make up a process description. However, we are also interested in how these activities interact with one another, what dependencies exist between these activities, and how these dependencies can be managed.

4.3.2 Dependencies and Resources
The Dependency Viewer places a great amount of emphasis on dependencies and elevates them to a first-class status equivalent to that of activities. Dependencies represent the interactions between various activities. Each dependency can potentially have several elements associated with it, namely producers, consumers, a resource, and a managing process. Often, dependencies represent the movement of resources from one part of the system to another. Thus, resources can be thought of as the objects or artifacts that are produced by certain activities and consumed by others, with dependencies being the mechanism by which resources are mediated. A producing entity is referred to as a producer and a consuming entity is referred to as a consumer. It is important to note that a maximum of one resource can be associated with a single dependency (A prerequisite constraint is an example of a dependency which might not have any resource associated with it). If there
are two distinct resources flowing between activities, then two separate dependencies
should be used to represent the two flows. It is possible, however, for composite resources
to be decomposed into subresources, in which case a composite resource could be associ-
ated with a dependency, where the producer creates the composite resource, and the parts
of the resource are consumed by multiple consumers.

A dependency can also have a managing process associated with it, that specifies the way
in which a dependency is managed. Managing processes are themselves activities and, as
such, they can be decomposed into a collection of subactivities and dependencies between
those subactivities.

Additionally, as will be fully discussed in Chapter 7, dependencies themselves can be
decomposed into finer dependencies, each of which can be managed by a separate manag-
ing process and can mediate a different resource. Finally, as with activities, dependencies
can have textual attributes, such as their name, to provide the supporting detail that is con-
sidered to be relevant to the representation.

4.3.3 Ports
Every activity can own any number of ports, each of which specifies a need for interaction
with other components of the system. Ports will be discussed more thoroughly in the fol-
lowing chapter, but it is important to note at this time that dependencies can only exist
between ports. Furthermore, it is not required that every port be associated with a depen-
dency from the very beginning, thus enabling the user to represent a particular activity in
isolation, specifying the activity's interface using ports independently of the dependencies
that will eventually connect to those ports.

Figure 4.1 gives an example, tying together some of these concepts and illustrating the
symbols used to represent each of the primitive objects. Activities are implemented as
rectangles, with adjoining square boxes on each activity representing the ports. The
dependency is displayed as a box, appropriately bordered by arrows depending on the
placement of the activities. An arrow pointing from an activity to the dependency describes a producer relationship, while an arrow pointing away from the dependency to an activity shows a consumer relationship. Lines connect ports of activities to arrows of dependencies and each line specifies a producer link or a consumer link, depending on the type of arrow to which it connects. Thus, in the figure, "Acquire Raw Material" produces a resource (i.e. the raw material) which needs to be transferred to "Process Raw Material." This transfer is represented by the "Transfer Raw Material" dependency. A line connects the port of the producer to the dependency (the producer link) and another line connects the dependency to the port of the consumer (the consumer link). If there were more than one producer or consumer, there would be more than one producer/consumer link connected to the appropriate arrow. Finally, we note that this transfer dependency is managed by a "Physical Transfer" mechanism.

![Diagram](image.png)

<table>
<thead>
<tr>
<th>Dependency Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name:</strong> Transfer Raw Material</td>
</tr>
<tr>
<td><strong>Resource:</strong> Raw Material</td>
</tr>
<tr>
<td><strong>Managing Process:</strong> Physical Transfer</td>
</tr>
</tbody>
</table>

**Figure 4.1:** Example showing symbols used to represent each of the primitive objects.

The following several chapters describe the main features of the Dependency Viewer and discuss why each one is important, how it enhances functionality, and how it is implemented. Chapter 5 discusses how an activity can be replaced by its decomposition while maintaining finest grain dependency rendering, and also further elaborates on ports. Chap-
ter 6 describes how a dependency can be replaced by its managing process, which can then be replaced by its decomposition, and so on. This mechanism of successive replacement, as well as its implications, are discussed. Finally, Chapter 7 shows how the viewer provides robust support for different dependency types and their associated managing processes, as well as for decomposable dependencies.
Chapter 5

Replacing an Activity with its Decomposition

5.1 Discussion

The first major feature of the Dependency Viewer is that it enables the user to replace an activity with its decomposition, while correctly rendering the dependencies at their most specific level. The power of abstraction is fully exploited here, allowing activities to be decomposed into a set of finer subactivities, each of which can be further decomposed and so on. While fully capturing the total information about the process, the editor also provides a convenient way of tailoring the amount of information that is displayed.

5.1.1 Abstraction

Using abstraction, we are able to represent a complex process by breaking it down into smaller manageable units. Clearly, if we tried to display ALL of the functional units that make up a process, we would have a highly cluttered display which would generate more confusion than insight. Our model is a top-down partitionable model in the sense that one can start from a high-level view of the process, and then focus in with detail on the particular area of interest. A good analogy is that of an atlas (Yourdon, 1989). An atlas first provides the user with a high-level map of the entire country showing the major components such as states, rivers, and major interstate highways. Subsequent pages of the atlas focus in on each state, devoting a whole page to one state, and showing a more refined level of detail which was not shown on the original high-level map, such as counties, local towns, and local highways. The same idea of abstraction applies to our process maps, which start off with a high-level view of the essential functional units of a process, and then allow the user to expand each of the major activities into their decompositions, which provide a successively finer level of detail about that particular activity.
Several important issues are raised by this abstraction mechanism, which are discussed in the following sections.

5.1.2 Decomposition Depth

One issue relates to how many levels we should have in our decomposition. This question can not really be rigidly answered in a fully generalized way and really depends on the level of functional detail in which the user is interested. We can say, however, that having a decomposition of more than roughly 6-10 subactivities is probably a good indication that one needs to group the subactivities into more abstract activities. Placing sixty subactivities in the decomposition of an activity is not a good idea, as it would undermine the advantages that could be gained by using abstraction.

5.1.3 Decomposition Focus

A related, although independent, issue has to do with the level of the decomposition at which we should be looking. Clearly, if a user is interested in what the steps of this particular process are, down to the operational level, then that user should look at the lowest level of decomposition to be able to fully understand exactly what is happening in this process. A different scenario is also feasible, where a group of consultants are brainstorming in order to come up with alternate mechanisms. In this case, it would make more sense to look at a higher-level decomposition, since this would not constrain the users into following this exact process, but would provide a framework for them from which to springboard. By remaining at a high level, the users could think of a variety of alternate ways that the activity could be executed.

5.1.4 Decomposition Consistency

There is also the issue of consistency between the parent activity and its subactivities, and this is the issue on which we will focus in this chapter. In our case, expanding an activity translates into replacing the activity with its decomposition. The decomposition of the activity will include its subactivities and the internal dependencies between these subactivities. Additionally, the decomposition provides information as to how the subactivities
map onto the external interface of the parent activity. That is, it determines which of the
subactivities connect to the parent activity’s external dependencies. It is only by correctly
and fully specifying this last part that we can achieve consistency and be able to properly
render dependencies at their finest level of specification.

5.2 Implementation

5.2.1 Ports and Dependencies

In order to implement this feature of finest grain dependency rendering, we added the port
object into the Process Handbook’s object set. As described earlier, a port specifies a need
for interaction between a particular activity and other activities. Thus, it describes the
external interface of this activity. For example, in Figure 5.1, there are three ports on the
left of the “Bake Cake” activity which describe the input resources which this activity
needs. The port on the right specifies the output resource that this activity produces. In
other words, this diagram says that cake mix, eggs, and milk are needed by the “Bake
Cake” activity, and that a cake is produced as a result of it. Similarly, Figure 5.1 also
shows the “Serve Cake” activity, with its associated ports, each of which needs a particu-
lar resource which the activity will consume, namely a cake, plates, and forks. To repre-
sent a relationship between these activities, we use a dependency, which connects the port
of one activity with the port of the other activity. In this case, the dependency is a flow
dependency, which represents the flow of the cake resource from the port of “Bake Cake”
to the port of “Serve Cake.”
Context: "Provide Cake"

Figure 5.1: Example showing decomposition of "Provide Cake".

This example illustrates several points. First, a dependency is never directly connected to an activity. Rather, a dependency connects the port of one activity with the port of another activity. Also, a port can have one and only one visible dependency associated with that port (what we mean by a visible dependency will be explained in the following chapter). This follows from the fact that a port specifies one specific interface, and is associated with at most one particular resource, and in fact will be connected to another port by a dependency, which we know can only be associated with at most one resource. To specify a different resource flow, separate ports would have to be created on each side of the flow and a separate dependency would represent that flow by connecting the newly created ports together.

It is also important to realize that the ports of an activity can be specified in isolation and they do not need to be associated with a dependency from the very beginning. As our example shows, the "Serve Cake" activity still has two ports which have not yet been "satisfied" and the "Bake Cake" activity still has three free ports!!! Of course, the presence of these free ports is an indication that the process description has not been fully specified as yet, but the flexibility of this editor is that it allows these partially specified process descriptions to exist. This flexibility is driven by the realization that people might want to leave a process partially specified or might be in the process of fully specifying it and want
to leave it in an incomplete state. Thus, by not forcing structural requirements on the user, a process entry mechanism is created that is flexible and not constraining. Additionally, we note that, having specified the ports of "Serve Cake", we did not have to satisfy the "cake" port by this particular flow dependency, but we could have placed the "Serve Cake" activity into a variety of contexts, constrained only by the fact that there would be SOME activity in that context which would produce the cake resource needed by the "cake" port.

In practice, ports can be created in two ways. If we have a clear picture of what the activity's interface should look like, as we did in the previous example, then the user can specify the ports of the activity without even thinking about how these ports will be satisfied. Often, however, activities are not analyzed in isolation, and there are a number of activities in the display whose interactions need to be represented by dependencies. In this case, the user can simply create a dependency between the activities, and the system automatically generates the ports on both ends of the dependency. These ports can then later be accordingly renamed to whatever the user prefers.

Finally, we note that, in our implementation, the ports of an activity are actually in the decomposition of that activity, and are differentiated from the other subactivities in the decomposition by the fact that they are port objects and not activity objects. This implementation decision has one main advantage. We have already said that when a new specialization of an activity is created, the attributes of that parent activity (including the decomposition) are automatically inherited by the new specialization. Thus, by placing the ports of an activity in the decomposition of that activity, the ports will also be automatically inherited whenever that activity is specialized, which is the behavior we desire.

5.2.2 Ports and Connectors

The second implementation tool that was introduced to enable the desired feature was that of connectors. There are two fundamental relationships that exist between activities in a decomposition graph, namely a peer relationship and a parent-child relationship. When
two activities are in the immediate decomposition of the same parent, we refer to them as peer activities. As we have seen, to represent an interaction between two peer activities, dependencies connect the ports of the two activities together. Similarly, we need to be able to specify the parent-child relationship, and that is how connectors are used. Continuing with our example, let us suppose that the “Bake Cake” activity is decomposed into two subactivities, “Mix Ingredients” and “Heat Mixture”. We have already created a flow dependency between the “Bake Cake” and “Serve Cake” activities. The problem is that, since we have not specified any parent-child connectors as yet, when we replace “Bake Cake” with its decomposition, we will not be able to capture the knowledge about the higher level flow dependency, thus losing essential information in the process of replacement. Of course, this is not the desired behavior.

The solution to the problem illustrated in the example is fairly simple. When we create the decomposition of an activity that already has ports, then we need to create a mapping between every port of the parent and one of the ports of the subactivities. Since the subactivities, when put together, form the parent activity, they must capture the total information about the parent activity, and, thus, it is essential that there exist such a one-to-one mapping between each of the parent’s ports and a port of its subactivities. As shown in Figure 5.2, we create an additional port of “Heat Mixture” and use a connector to link this new port with the “cake” port of “Bake Cake”. Now when we expand “Bake Cake”, the system will trace the connector relationship and correctly render the display, as shown in Figure 5.3, where we notice that it is really the “Heat Mixture” activity which produces the flow dependency at this finer level of detail.
Of course, in order to create a fully consistent process description, we would also need to connect each of the input ports of “Bake Cake” to corresponding input ports on “Mix Ingredients.” In this case, we didn’t notice their absence because the parent’s ports were not connected to any dependency. However, to COMPLETELY represent a decomposition, we require that every port of the parent be mapped onto a port of one of its subactivities, with a connector creating the link between the ports. Again, we notice that the system is able to support process descriptions in a partially specified state, which points again to the flexible nature of the editor and the power which it yields to the user in deciding when and how to complete the description.

**Figure 5.2:** Connectors.

**Figure 5.3:** Result of replacing the “Bake Cake” activity with its decomposition.
At this point, it is important to differentiate between dependencies and connectors. As we have already stated, a dependency connects a port of one activity with a port of a second peer activity. Similarly, a connector creates a mapping between a port of a parent activity and a port of one of the parent’s subactivities. Connectors are fundamentally different and represent a much simpler relationship than do dependencies. For example, connectors are not associated with any resource, nor can they be managed. They are simply the mechanism by which we can specify the relationship between a parent’s port and its child’s port. However, they do share the structural requirement with dependencies that only one connector can be associated with a single port. This follows from the fact that the role of the connector is only to map the parent’s port to a child’s port, and since the parent’s port is connected to only one dependency, only one port among the subactivities should be linked with that parent’s port, and hence there should only be one connector associated with every port. Clearly, however, we see that a single port will often be associated with both a dependency and a connector, as Figure 5.2 shows.

5.2.3 Bottom-Up Specification and System Generation of Connectors

So far, we have been assuming that the dependencies will be specified in a top-down fashion. In the example we presented, we showed that the flow dependency was first created between the peer activities, and then, at some later point, the user decided to connect the parent/child ports using connectors to capture the information about where to connect that flow dependency when we replace the parent activity with its decomposition. Of course, it is entirely possible that the bottom-up approach may be used to specify the dependency. In this scenario, the user would have first specified the activities and the subactivities of each activity and saved the dependencies for last. At this point, if the user initially creates a dependency between two activities that are not peers, then the system automatically generates an appropriate set of connectors and a peer level dependency to represent the new dependency. For example, in Figure 5.4.a, let us assume that the decomposition structure shown has been specified, but there have been no dependencies created. “A” and “B” are children of the same parent “X”. “A” can be decomposed into “AA”, which can further be
decomposed into “AAA”. Similarly, “B” has “BB” in its decomposition. At this point, the user decides to create a dependency between “AAA” and “BB”.

**Figure 5.4:** (a) Decomposition structure of bottom-up specification scenario. (b) Result after creating dependency between “AAA” and “BB”.

In order to represent all dependencies in a consistent manner, the system will automatically trace the lineage of both activities until it comes across a peer-to-peer relationship, appropriately creating ports along the way, with connectors linking parents to children and with the actual dependency being created only at the peer-to-peer level. The resulting structure created by the system is shown in Figure 5.4.b. A port is automatically created on “AAA” and linked to another created port on “AA” by a connector. Then, another connector is used to link the port on “AA” with a newly created port on “A”. The dependency is created only between this port of “A” and a newly created peer port of “B”. Finally, a third connector is created between this port of “B” and a newly created port of “BB”. In order to render a non-peer relationship, such as the one between “AAA” and “BB”, the system executes a routing algorithm which finds all the connectors linked to the ports of the peer-level dependency and follows the connectors on each side of the dependency
down to the activity that is currently visible on the display. It then renders the dependency between “AAA” and “BB”, leaving the implementation details largely hidden from the user.

Thus, finest grain dependency rendering is implemented by breaking down every relationship into a set of zero or more connectors and a single peer-level dependency. Actual dependencies can exist only between peer-level activities, with connectors linking parents and children to represent non-peer relationships. Clearly, allowing the system to generate all of these connectors is a much more efficient way of creating dependencies than forcing the user to connect the ports manually, as they would have to do if they initially created a high-level dependency. Thus, whenever possible, it is recommended that dependencies be created at the lowest level possible, so that the system can facilitate the entry of dependencies and so that manual connection by the user is minimized.

5.2.4 Context and Multiple Dependency Viewers

The idea of context is fundamentally important in correctly specifying an activity’s interface. The visual metaphor that the dependency viewer uses for the relationship between an activity and its subactivities is that of a large “parent activity” box containing several smaller “subactivity” boxes, each of which may contain several even smaller “subactivity” boxes and so on. Thus, in our mind, we could represent our original example as shown in Figure 5.5. We see that the ports of the subactivities (smaller boxes) that are inside the parent activities (larger boxes) are appropriately connected to the ports on the edges of the larger boxes, as we had specified that they should be. This is a concise and accurate way of representing how the ports of a parent activity are connected to the ports of its subactivities. We say that the smaller subactivities are within the context of the larger parent activity. Similarly, the two main activities “Bake Cake” and “Serve Cake” must themselves be subactivities of some other activity (which we would call the root of this decomposition). The root activity might be called “Provide Cake” in our case, which would have “Bake Cake” and “Serve Cake” in its decomposition.
Before opening the Dependency Viewer, the user needs to first select a root activity whose decomposition they wish to examine and which will serve as the overall context of the display. The viewer will then be called, displaying the subactivities of that root activity. Thus, in our example, we would have opened the Dependency Viewer within the context of "Provide Cake" and the resulting initial display is that of Figure 5.1. Now, at this point, the user has two options. They can replace any of the subactivities of the root activity with their decompositions keeping the other subactivities in the display (i.e. replace "Bake Cake" with its decomposition while keeping "Serve Cake" in view, as Figure 5.3 shows). Alternately, they can decide that they are really interested in focusing in on the decomposition of one of the subactivities without preserving the peripheral context, in which case they would select the subactivity of interest and call the dependency viewer again. If we did this in our example, deciding that we want to focus in on "Bake Cake" alone, a new window would be created with an initial display as shown in Figure 5.6. As we can see, the display shows not only the ports and dependencies associated with the subactivities but also shows the ports of the parent process on the perimeter of the screen linked by connectors to the ports that we had earlier specified. Thus, the box-within-a-box metaphor is clearly illustrated.
The reason that multiple dependency viewers are crucially important in our design is that they allow the user to manually connect the ports of a parent to its child. Since we always replace an activity with its decomposition, we never have the parent activities and its children on the screen at the same time. As Figure 5.6 shows, the sole exception to this is with the root activity and its subactivities, which is the only case where the ports of the parent and the ports of the children are both on the screen simultaneously. Thus, to manually connect the ports of a parent and its child, the user needs to open a new window with the parent as the root and can then connect the ports appropriately. Again, we reiterate that this manual connection exercise can be minimized if the user specifies the dependency at the lowest level possible, enabling the system to automatically generate the connectors appropriately.
Chapter 6

Replacing a Dependency with its Managing Process

6.1 Discussion

As mentioned earlier, coordination can be thought of as ways of managing dependencies between activities. A major distinguishing feature of the Process Handbook methodology is that there is a distinction made between a dependency and the process that manages it. This distinction is fundamentally important in order to separate the specific surface structure of a business process from its more fundamental deep structure (Malone, 1996).

6.1.1 Deep Structure vs. Surface Structure

The deep structure of a business process corresponds to the core activities of that process and the dependencies between these core activities. The deep structure captures the fundamental essence of the process, but leaves some of the details having to do with the specific execution of that process unspecified. Specifically, the deep structure does not specify how each of the dependencies are managed, leaving open a variety of alternate ways that each dependency could be managed.

The surface structure of a business process fills in the coordination details that the deep structure leaves out. Rather than representing interactions between the core activities by high level dependencies, the surface structure represents these interactions by the actual managing process that coordinates these dependencies. Thus, a surface structure of a business process is like a deep structure with all the dependencies replaced by specific processes to manage them. Since each dependency can potentially be managed in a number of different ways, a particular surface structure corresponds to one particular way in which the dependencies can be managed. Thus, there can be several different surface structures for the same deep structure. This trade-off in representation is similar to that between granularity and precision. If we represent the activity using a large-grained definition, then
the activity can be executed in a variety of ways, and we are left unconstrained, with all the possibilities and permutations before us. As we get more precise (surface structure), we focus in on a specific way that this activity is actually executed. There can be actually be several levels of surface structure, but we limit ourselves to only one level for purposes of this discussion.

![Diagram](image)

**Figure 6.1:** Deep structure of a production process. Taken from (Malone, 1996).

### 6.1.2 Example

Let us illustrate these concepts. Figure 6.1 shows the deep structure of a production process. The two core activities are “Make Tires” and “Make Cars” and there is a flow dependency connecting their ports together. In order to focus in on the new concepts of this chapter, ports are not drawn on the diagrams in this discussion section. Although this is the essence of the process, we need a particular surface structure to specify the details of how that flow dependency is managed. Flows can be managed in a variety of ways. To manage the flow of the tires from “Make Tires” to “Make Cars”, we can imagine two separate coordination strategies. One way would be to use the popular “Make to Inventory” scheme where tires would be produced to keep the inventory full, and then tires would be removed from that inventory as orders came in. Figure 6.2 shows the surface structure of this production process where the flow dependency has been replaced by its “Make to Inventory” managing process. A second method, “Make to Order”, would dictate that tires should only be produced after an order has arrived that demands that a certain number of tires be produced. This second surface structure is shown in Figure 6.3. Clearly, there are advantages and disadvantages to each approach. While the first surface structure using
"Make to Inventory" to manage the flow would have a very fast response time to an order, there would also be significant inventory costs. Conversely, the second surface structure would have low inventory costs, but would not be able to respond to orders very quickly. Thus, we see that the same deep structure has different surface structures depending on how we choose to manage the dependencies of the deep structure.

**Figure 6.2:** Surface structure 1: Make to inventory. Taken from (Malone, 1996).

**Figure 6.3:** Surface structure 2: Make to order. Taken from (Malone, 1996).
We have seen that a dependency on a deep structure diagram is really an abbreviation for the coordination process that is managing that interaction. The dependency serves as an abstraction grouping which concisely represents the details of the actual managing process (which includes the activities that make up the managing process, as well as the dependencies between those activities) in the form of a single dependency. Clearly, this makes for much less cluttered diagrams, while still capturing the information needed to specify the particular surface structure.

6.1.3 Motivation for Implementation

In the Process Handbook methodology, a business process is represented by its deep structure and then each dependency is associated with one of several appropriate managing processes, creating a particular surface structure. Initially, only the deep structure of a business process is displayed. Clearly, we would like to be able to get at the specific surface structure.

This leads us to the motivation for implementing this feature of replacing a dependency with its managing process. One driving force is simply to enable the user to reveal the surface structure of the business process being studied. By carrying out this replacement, the user is able to see the particular ways in which each of the dependencies are being managed. Additionally, this feature also reveals the underlying dependencies between the activities of the managing process which were not visible in the deep structure. This is important because it may be that some of these dependencies are not being managed effectively and that may turn out to be the reason that the process is not executing efficiently. In our previous example of the "Make to Inventory" surface structure, there is a prerequisite constraint between two of the activities of the managing process, namely "Order tires" and "Remove tires from inventory". This prerequisite constraint needs to be effectively managed to ensure that tires are ordered before removing the appropriate number of tires from our stock. By revealing the surface structure, we may discover that this prerequisite constraint is not being managed effectively, and could be managed using a more efficient pro-
cess, or we may find that it is not being managed at all. Thus, this dependency replacement feature enables the user to view the components of the managing process, both its activities and its dependencies, which are otherwise hidden from the user in order to have a more concise representation.

In order to derive more usefulness out of this feature, we notice that each activity of the managing process may, itself, be decomposable into finer functional activities and dependencies. If we combine this dependency replacement feature with the decomposition replacement feature from the previous chapter, we come up with a composite feature of successive replacement. This allows users to replace a dependency by its managing process, revealing a set of underlying activities and dependencies. Each of the activities of the managing process can then be replaced by their decompositions and each of the dependencies can be further replaced by their managing process, and this cycle of successive replacement can continue as long as the necessary information has been specified. Thus, the user can really zoom in to great depth on a particular activity or dependency of interest, while at the same time keeping all of the other original high-level activities and dependencies in the peripheral context.

Going to our previous example, again, we can imagine that the “Store tires” activity can be decomposed into the two subactivities “Open Warehouse” and “Place Tires in Warehouse”. Thus, using successive replacement, the user can go from the original display of Figure 6.1 to the display of Figure 6.4 by replacing the flow dependency by its “Make to Inventory” managing process and then by replacing the “Store tires” activity by its decomposition.
6.2 Implementation

In order to implement this dependency replacement feature, we notice that the dependency itself is actually just an abstraction for the managing process that manages it. The dependency is simply a concise abbreviation for the actual surface structure of the process that is being studied. Thus, what we would like is for the ports that form the endpoints of a dependency to also somehow connect to the managing process activity that manages that dependency. This is exactly how this feature is implemented.
Figure 6.5: System generation of managing dependencies after setting managing process.

When the user decides to manage a dependency by setting a coordination activity as its managing process, the system automatically generates two new dependencies. One dependency connects the dependency’s producer port with a newly created consumer port of the managing activity, and the second dependency connects a second newly created producer port of the managing activity with the original dependency’s consumer port. This behavior is illustrated in Figure 6.5, where the original flow dependency connects ports P1 and P2, and the two new system-created dependencies connect ports P1 to P3, and ports P4 to P2. Of course, this is only a representation of what the system is doing behind the scenes, and this is not what would be rendered on the display. In order for this scheme to work, we require that, at any given time, either a dependency is visible or its managing process is visible. Both should not be visible at the same time. Clearly, this requirement is always fulfilled by definition. We will either be representing the interaction by the dependency, or we will want to expand the dependency to reveal the surface structure by replacing it with its managing process.

Here, we see that this is the one exception to the rule which requires that there be only one dependency associated with a particular port. Clearly, in this case, port P1 and port P2 are each associated with two separate dependencies. It does not upset the rendering algo-
rithms, however, because only one of their associated dependencies will ever be visible at a given time. Thus, a more precise rule for the number of dependencies that can be associated with any one port is that a port can only be associated with one VISIBLE dependency on the display at any given time. As is clearly evident from Figure 6.5, a port CAN be associated with more than one dependency, where one is a peer-level dependency ("flow"), and the other is a dependency to the activity ("Make to Inventory") that manages that peer-level dependency.

![Diagram](image)

**Figure 6.6:** The connector relationships between the external ports of "Make to inventory" and the ports of its subactivities.

In general, the managing process will usually have a decomposition. Thus, having set the managing process for a dependency, we would then proceed to connect the newly created ports of the managing process to the ports of its subactivities in order to enable the successive replacement feature discussed earlier. For example, Figure 6.6 shows the connector relationships between the external ports of "Make to Inventory" (P3 and P4) and the ports of its subactivities. After these connections have been specified, then the user would be
able to successively replace the "flow" dependency by its managing process and then replace the managing process by its decomposition, rendering the display as already shown in Figure 6.2.

To fully specify a process description so that the steps of the process can be executed, it is clear that every dependency that exists in the process map needs to be managed by an appropriate coordination process. Having only the deep structure does leave us unconstrained to explore several alternative ways of managing the dependencies. However, if we are representing a particular process, then we will always want to specify what specific coordination processes are actually used to manage the dependencies in that particular process. Without associating every dependency with a process to manage it, we would be left with a partially-specified representation. Having said that, as before, we recognize the flexibility gained from allowing the user to leave representations in a partially-specified state. With that in mind, the system does not place any structural constraints on the user which would force them to set the managing process for every dependency on their process map.
Chapter 7

Robust Support for Dependencies

7.1 Discussion

Whereas previous viewers have emphasized the inheritance relationships between activities, the Dependency Viewer stresses the dependency relationships by providing robust support for creating, manipulating, and managing these dependencies. Previous viewers supported only a single generic dependency type, forcing the user to visually differentiate between the various types depending on the context. The Dependency Viewer provides a more robust mechanism where an extendable variety of dependency types can be specified and rendered. Additionally, just like activities, dependencies can now be decomposed and viewed as collections of finer subdependencies.

By providing support for an extendable list of dependency types, the viewer is enabling the user to use a richer abstraction language when specifying the process description. Different types of dependencies can represent fundamentally different interactions between activities, thus allowing the user to correctly express a broad and important class of interactions which are inadequately represented without this richer language. Additionally, by distinguishing between different dependency types, we can also then select appropriate coordination mechanisms depending on the dependency type.

7.1.1 Dependency Types

Although there are many different kinds of interactions which can take place between activities, three major dependency types have been identified. Each is thought to be a fundamental dependency type that needs to be distinguished from the other types to enable a richer representation and to ensure that it is appropriately managed. The three dependency types are flow, sharing, and fit dependencies, each of which will be briefly discussed.
A flow dependency is perhaps the most common dependency type and represents the producer/consumer relationship which exists between an activity which creates some usable resource and an activity which uses that resource. To illustrate its usage, we refer back to Figure 6.1, where a flow dependency exists between the “Make Tires” and “Make Cars” activity. In this case, the resource associated with the flow is the tires. A flow dependency can be decomposed into three more primitive dependencies: prerequisite, transfer, and usability dependencies. To better understand a flow dependency, we examine each of its parts. Flows have a prerequisite component, which dictates the requirement that the resource has to be produced before it can be consumed. In our case, we have to make sure that the “Make Tires” activity completes execution before the “Make Cars” activity is initiated. As we have already seen, this dependency could be managed by a “Make to Order” process or by a “Make to Inventory” process, each having its advantages and disadvantages. The actual transfer of a resource is also captured by a flow dependency. For example, we will need some mechanism to actually move the tires from the producer activity to the consumer activity. This dependency might be managed by a “Physical Transport” process, in which case it would require some actor to actually move the resource from one activity’s site to the other. Alternately, a “Make at Point of Use” process might be employed, where the tires would actually be produced in the same place where they would be used (i.e. the automobile assembly plant). Finally, flow dependencies also include the usability aspect of the flow. We have to make sure that the resource is delivered by the producer in a form that is usable to the consumer. In this example, we have to ensure that the tires are of the right radius, thickness, and texture to fit onto the axle of the particular model for which they have been produced. This usability constraint is often handled by a “Use Standard” process or an “Ask Customer” process.

A second fundamentally different dependency is the sharing dependency which exists whenever there is a limited resource which needs to be shared by multiple activities or actors. This resource could be some physical resource, such as a computer needed by many developers, or it could be a more abstract resource that needs to be shared, such as time or space. An example of this would be many students asking for help from a teaching assistant (T.A.). In this case, the T.A.’s time would be the limited resource, and a process
would be needed to manage this dependency between the students and the T.A. Since resource allocation is a common task that needs to be performed in a variety of fields, it has been extensively studied, and there are several coordination processes that can manage a shared resource dependency, including “first come/first serve”, priority order, managerial decision, budgets, and market-like bidding. Continuing with our example, in order to manage the limited time resource, the T.A. could employ a “first come/first serve” approach in which he would field questions in the order they were received. Alternately, the T.A. could use a budgeting mechanism whereby each student would have an assigned time to ask their questions. It is possible that the T.A. may decide to use market-like bidding, helping whoever is willing to pay the most, or perhaps arbitrarily choose which student to help (managerial decision).

Finally, the third major type of dependency is the fit dependency, which represents the relationship which has to exist when two or more activities are jointly creating a resource. An example would be several software programmers developing the same software application. In this case, the programmers have to coordinate their efforts to make sure that they are coding their parts in a way that will be compatible with the other parts and that the final product will meet the desired specifications. The fit dependency can be managed by processes such as “Total Simulation” used by Boeing or the “Daily Build” mechanism used by Microsoft.
<table>
<thead>
<tr>
<th>Dependency</th>
<th>Example Coordination Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow</td>
<td></td>
</tr>
<tr>
<td>Prerequisites</td>
<td>Make to order vs. make to inventory</td>
</tr>
<tr>
<td>Transfer</td>
<td>Transport vs. make at point of use</td>
</tr>
<tr>
<td>Usability</td>
<td>Ask customer vs. use standard</td>
</tr>
<tr>
<td>Sharing</td>
<td>First come/First serve, priority order, budgets, managerial decision, market-like bidding</td>
</tr>
<tr>
<td>Fit</td>
<td>Boeing’s total simulation vs. Microsoft’s daily build</td>
</tr>
</tbody>
</table>

**Figure 7.1:** The basic dependency types and example coordination processes.
Taken from (Malone, 1996).

Figure 7.1 summarizes all the basic dependency types and gives example coordination processes for each of the types. We note that each of these dependency types can be further specialized when applied to particular scenarios. For example, we might want to create alternate sharing dependency specializations based on several dimensions of its resource classification (i.e. whether or not the resource is perishable, divisible, physical, etc.). Thus, a dependency specialization hierarchy could be established.

### 7.1.2 Managing Processes

The managing processes that we have been describing are really activities just like any other activity, except that they are associated with a particular dependency. Thus, they can also be decomposed into a set of subactivities and dependencies. In fact, by developing process representations for each of the coordination processes, we should be able to concisely represent much of coordination activity as specializations of these generic processes, thus creating a coordination process hierarchy with a direct mapping to the dependency hierarchy.
Often, in reality, it is easier to see the managing process which exists since that is actually the tangible process which can be identified by conducting field studies and interviews. Why then don’t we simply include this coordination process and its subactivities in the list of subactivities that make up our process? There are two main reasons for abstracting these coordination processes into common dependency types. As we have already stated, the first has to do with being able to concisely represent processes. If we literally included every single coordination mechanism present in a process, then our diagram would become very cluttered very fast. But there is a more powerful concept at play here, and that is one of generativity. By classifying dependencies into a specialization hierarchy, we can group similar types of dependencies together. Having grouped the dependencies, each dependency class can then be associated with a coordination mechanism. As future dependencies are identified as belonging to this particular dependency class, their associated coordination processes can then also be added to the list of processes capable of managing this type of dependency. If we are able to then specify a process as a group of activities interrelated to each other by dependencies belonging to dependency classes in our hierarchy, then the system can automatically generate a list of managing processes which can be used to manage these dependencies. This would not require any knowledge of the actual process, but would only use the fact that this particular dependency type is being used in this process. In this way, users can be presented with alternative mechanisms to coordinate dependencies. The hope is that some of these alternatives may be non-obvious choices which the user would not have thought about, but that the system is able to generate because it has a set of coordination processes associated with each dependency type.

Thus, we can already see how the Handbook might provide opportunities for innovation. Focusing in on a transfer dependency, let us suppose that the system remembers that a transfer dependency may be managed by a “Physical Transport” mechanism or by “Make at Point of Use.” If the user is then examining the process by which a credit card company disburses bills to its customers, they may represent this relationship using a transfer dependency, assuming that it would need to be mailed to the customers in an envelope as is traditionally done. When they subsequently try to set the managing process for this dependency, however, the system would also generate the “Make at Point of Use” possi-
bility, which may allow the user to think up new ways of executing this process such as having the bills printed directly at the customer’s home or faxed directly to their fax machines.

7.1.3 Motivation for Decomposable Dependencies

The advantages of being able to decompose dependencies should be fairly clear by now, since we have exploited the power of abstraction so many times in this project. Basically, we are taking advantage of the same leverage that we gained by being able to decompose activities into subactivities. By decomposing dependencies into finer subdependencies, we are creating an even richer language which can support dependencies in an even more robust and concise manner. The advantage of conciseness is evident from our specification of a flow dependency as being composed of three more primitive dependencies. Rather than forcing the user to display all three dependencies whenever they want to represent a flow, the system enables them to group the three subdependencies into a larger flow dependency and render the relationship with a single dependency, which can be expanded to reveal the three underlying dependencies. Each of these dependencies might be further decomposed into finer dependencies. Thus, decomposable dependencies enable the user to decide what level of detail they wish to view. A user might be particularly interested in one particular dependency in a process map, and so they could expand that dependency to see its decomposition, while leaving the other high level dependencies unexpanded in their original form. Additionally, if we had simply left the flow dependency without any decomposition, then we would not have been able to distinguish the three components that make up that flow, and, subsequently, we would not have been able to identify what types of coordination processes should be used to effectively manage each of the components.

7.2 Implementation

In order to provide support for different dependency types, there are two main implementation design issues. The first has to do with how the user would specify a dependency,
while the second is involved with how the dependency would actually be visually rendered on the display.

7.2.1 Dependency Specification and Rendering

In order to introduce a dependency onto a process diagram, the user needs to select an entity (either an activity or a port) and then choose the “Create Dependency” option from the menu. At this time, the system displays the dependency specialization hierarchy that was referred to earlier. This hierarchy contains the major dependency types and their specializations (if any) and can be arbitrarily extended by the user. Each dependency is also typed to distinguish it from the others and so that an appropriate set of coordination mechanisms can later be associated with it. Having specified the dependency type, the user then needs to select a second entity (again, either an activity or port), and a dependency between the two entities is then generated. It is important to note that there are different semantics for what the first and second entities represent depending on what dependency type is specified. For a flow dependency, the first selected entity represents the producer while the second represents the consumer. For a sharing dependency, on the other hand, both entities are considered to be consumers. Conversely, a fit dependency constrains both entities to be producers. When the user wants to manage a dependency, they can choose the “Set Managing Process” menu option after having selected the dependency. The system will then generate a list of appropriate coordination processes which can be used to manage this process. This list can also be arbitrarily extended by the user as they see fit.

The reason we keep mentioning that either activities or ports can be the selected entities is that the system provides support for creating dependencies between both. Of course, dependencies can only exist between ports. Recognizing the fact that the user may often want to specify a dependency just between activities, the system checks the entity object types that are selected. If it is a port object, as it would be for the manual connection of ports, for example, then the system need not do anything extra. However, if an activity object has been selected as either endpoint, then the system automatically creates a port on that activity and connects the dependency to that newly created port.
Figure 7.2: A fully generalized dependency label with a producer and consumer arrow on each face.

In order to render the different dependency types, the main visual feature of distinction is the appropriate use of arrows. As mentioned earlier, the dependency itself is represented by a square box, with a set of arrows around its border, depending on whether the activities are producers or consumers, and also depending on their relative placement with respect to the dependency label. Arrows pointing into the dependency label represent producers while arrows pointing away represent consumers. Since the system leaves the user unconstrained as to where they would like to place the activities, it is possible in the fully generalized case, that there may be a producer and consumer above, below, to the right, and to the left, of the dependency label. Thus, a fully generalized dependency label would look like Figure 7.2, with a producer and consumer arrow on each face of the label to accommodate user flexibility in activity placement. If there are many producers on any particular face, several links connect the ports of these producers to the producer arrow for that face, and likewise for consumers. Since these labels can become fairly large and bulky, and since the normal case is to have only two arrows, the system automatically determines how many arrows need to be used, and accordingly shrinks the dependency label to its most manageable size. Some typical dependencies are shown in Figure 7.3.
Figure 7.3: Graphical representation of typical dependencies.

In order to specify multiple producers or consumers for a dependency, the user needs to append the desired entity to an already existing dependency. Thus, to add a producer to a dependency, the user would select the new producer they wish to add, choose the "Create Dependency" menu option, and select the dependency label itself for their second selection. Similarly, to add a consumer, the user's first selection would need to be the dependency label and their second selection would be the new consumer they wish to add. A multi-producer/consumer flow dependency in the context of a verification process is shown in Figure 7.4, which illustrates the additional representational feature of decomposable resources. In this case, the resource associated with the dependency is the entire customer record taken from a database. This resource, however, can be decomposed into several fields, such as the name field, social security number field, date of birth field, etc. Thus, the consumers of this dependency do not necessarily need to receive the entire resource but can receive part of the decomposition of the resource. The figure shows that each "Verify" activity gets the appropriate subfield of the entire record.
Figure 7.4: Example of a flow dependency with one producer and two consumers. This figure also illustrates the use of decomposable resources.

There is a fair degree of complexity involved with adding a producer or consumer to a dependency because of the previous features that we need to also support. For example, adding a non-peer producer to a dependency leads to the same system-generation of connectors that would be needed to support our finest grain dependency rendering feature (Our earlier example in Figure 5.4 illustrates this process). Additionally, if the dependency to which a new producer or consumer is being added is already managed, then the new entity needs to also be added as a producer or consumer to the extra managing dependencies that connect to the managing process for the dependency (See Figure 6.5 for a description of these managing dependencies).

7.2.2 Decomposable Dependencies

Finally, we come to the implementation of decomposable dependencies. We used two new tools in order to support this feature, namely subconnectors and decomposable ports. In this new scheme, a parent port object can contain several subports in its decomposition. The dependencies connected to these subports can then be thought of as being decomposi-
tions of the parent dependency connected to the parent ports. Figure 7.5 gives an example to illustrate this concept.

![Diagram](image)

**Figure 7.5:** Decomposable dependencies implemented using decomposable ports.

The figure shows two parent ports, each of which contain three subports in their decomposition. Port P1 can be decomposed into subports P2, P3, and P4 while the other parent port (P5) decomposes into subports P6, P7, and P8. By having the four dependencies connected to each of these ports as shown, we can effectively represent the flow dependency as being decomposable into the more primitive prerequisite, transfer, and usability dependencies. When the user wishes to replace a dependency by its decomposition, the system simply expands the ports that are connected to that dependency revealing the subports in their decompositions and rendering the subdependencies which are connected to these subports. Thus, a dependency can be replaced by its decomposition, removing the ports of the original dependency, and replacing them with their subports that are connected by the subdependencies. Of course, the reverse process of collapsing back up to the parent dependency is also supported. Just as there is an advantage gained from having ports be in
the decomposition of their associated activities, so too does placing the subports in the
decomposition of their parent ports facilitate decomposition inheritance.

As before, we needed to make a few additional modifications so that this feature is com-
patible with the earlier feature of activity replacement. In order to maintain the connector
routing algorithms used for finest grain dependency rendering, we introduce a new kind of
relationship called a subconnector, which connects a subport in the decomposition of a
parent activity's port with a subport in the decomposition of a child activity's port. It is
very similar to a connector, but it is useful to distinguish a connector from a subconnector
so that the rendering algorithm can execute more efficiently. Thus, when a dependency
needs to be rendered, the algorithm again takes the two endpoints of the dependency and
traces down the connectors from each endpoint until it finds a connection to an activity that
is visible. This was how the original algorithm worked. To deal with the additional dimen-
sion of decomposable ports, a leaf attribute is associated with each port which determines
whether the port has been expanded to reveal its subports (in which case the parent port
would not be a leaf while the subports would each be leaves). Thus, the algorithm can con-
tinue down the port decomposition hierarchy until it finds those ports which are leaves and
can then render the dependencies which are connected to those ports.
Figure 7.6: Placing the transfer dependency in the decomposition of the flow dependency.
(a) The state of the process map before executing this action.
(b) The state of the process map after execution.

In order to specify that a subordinate dependency should be in the decomposition of a more general dependency, the user needs to create both dependencies and then drag the subordinate dependency and drop it over its parent dependency. By doing this, the system removes the ports that were connected to the subordinate dependency from the activity where they were specified and appropriately places them as subports in the decomposition of the producer and consumer ports of the parent dependency. This process is illustrated in Figure 7.6 in a scenario representing the state of the process map before and after the “transfer” dependency is placed in the decomposition of the “flow” dependency. In Figure
7.6.a, both of the dependencies have been created as going from “A” to “B”. At this point, we notice that “A” has a subactivity “AA” in its decomposition, and there is a connector linking P5 to P1, meaning that “AA” is the subactivity which is related to the “flow” dependency. Now, let us imagine that the user drags and drops the subordinate “transfer” dependency over the “flow” dependency, in order to specify that the “flow” dependency can be decomposed into a “transfer” dependency. Figure 7.6.b shows the resulting state after executing this drag-drop operation. We notice that P3 and P4, the ports associated with the “transfer” dependency, have been removed from the decomposition of “A” and “B” and have been placed in the decomposition of “P1” and “P2”, respectively. Additionally, since there is already a connector relationship between P5 and P1, the system automatically generates a new port (P6) to maintain the required mappings. P6 is placed in the decomposition of P5 and is connected to P3 by a subconnector, whereas P5 is still connected to P1 by a connector. Although we have used a simple example to illustrate this process, the system would automatically create new subports and subconnectors for more complicated scenarios as well.
Chapter 8

Future Work

There are several additional features which could be incorporated into the dependency viewer to increase its functionality, including support for checking port compatibility, several enhancements to the user interface, and a dependency management tool.

One additional area of future work is in the area of port compatibility. As we have seen, ports specify a specific need for interaction, usually requiring a particular resource to satisfy this need. Currently, this need for interaction is not stored by the system in any way. The system simply connects ports with dependencies, and does not check to see if the ports that are being connected are compatible with each other or if this dependency type is an appropriate choice to satisfy the needs of these ports. By encoding each port with certain attributes and developing a consistent resource-naming schema, the system could automatically check for port compatibility when creating dependencies. For example, if the user had already specified the ports of two activities in isolation, and both ports needed a "cake" resource, then, when the user tries connecting these ports with a dependency, it would be useful to have the system make sure that both ports are compatible (i.e. whether or not they need the same resource).

There are also several interface enhancements which would increase user functionality. While the system does allow users to move activities and persistently saves the activity’s coordinates, additionally allowing users to resize activities would allow for an even more flexible display. Further enhancements include allowing the user to move and place the dependency labels as they choose and allowing ports to be movable objects around the perimeter of the activity with which they are associated. There is also room for improvement in the interface that specifies how a user can manually connect a parent’s port to its child’s port. Finally, an optional automatic layout option could use some graph algorithm to layout the process description in an efficient manner, minimizing lines from crossing
and boxes from overlapping. In this way, the user would have the flexibility of choosing to layout the process themselves or have the simplicity of having the system generate an efficient layout, which could then be further customized by the user.

Other future work in this area that would be helpful would be to create a dependency management tool that would detect and automatically generate a “work item” in order to indicate what steps the user needs to execute to fully specify the process representation. Every time a dependency was created, a “Set managing process for dependency x” would be added to the list. Additionally, the system could detect whether there were any partially specified decompositions, meaning whether there was always a mapping from the ports of a parent to some port in its decomposition (assuming the parent had a decomposition). If a mapping did not exist, a “Connect port y of activity z to some port in its decomposition” work item would be appended to the list. In this way, the user would be able to have the flexibility to leave processes partially specified, while also having a persistent record that would remind them what needs to be done in order to complete the specification. As the user executed each of the activities, the corresponding work item would be removed from the list. Taken a step further, the user could also click on each work item and have the system automatically open the appropriate windows and screens and guide the user through the editing process, leading them through a series of steps to ensure that their process description is completely specified.
Chapter 9

Conclusion

As opposed to earlier viewers that stressed the inheritance relationships, the Dependency Viewer places emphasis on the dependency structure and the processes used to manage them. Additionally, the system provides a more flexible layout mechanism, allowing users to specify a process description using a more familiar two-dimensional process map paradigm. There are several main features that this viewer enabled, many of which exploited the powerful concept of abstraction, which is a major element of the Process Handbook methodology.

The system allows the user to replace an activity with its decomposition, while maintaining finest grain dependency rendering. By providing an abstraction mechanism, the system is able to capture full information about the process, while simultaneously providing the user with a convenient way of tailoring the amount of information that is displayed.

This feature was implemented by introducing the port object to our schema, as well as by employing connectors. Finest grain dependency rendering was enabled by breaking down every relationship into a set of zero or more connectors and a single peer-level dependency. It was recommended that dependencies be created at the lowest level possible, so that the system can facilitate the entry of dependencies by automatically generating connectors and so that manual connection by the user is minimized.

The second main feature provides the user with the capability of replacing a dependency with its managing process. One motivation for implementing this feature is that it enables the surface structure of a business process to be revealed, allowing the user to see the ways in which each dependency is being managed. Furthermore, this feature also exposes the underlying dependencies and subactivities of the managing process, which were hidden from the top level view. Finally, by combining this dependency replacement feature with the decomposition replacement feature, the user can use successive replacement to zoom
in on a particular area of interest, while keeping the other high-level activities and dependencies in the peripheral context. Managing dependencies automatically generated by the system were the main tool used to implement this feature.

Finally, the system provides support for specifying and rendering an extendable list of dependency types and decomposable dependencies. This provides the user with a richer abstraction language with which to specify process descriptions. Furthermore, it allows appropriate coordination mechanisms to be selected depending on the dependency type. By allowing dependencies to be decomposable, the system allows dependencies to be composed of finer subdependencies, yielding a more concise high-level representation, which can be expanded to reveal the underlying details. In order to implement this feature, subconnectors and decomposable ports were employed.

The viewer provides flexibility by enabling users to arbitrarily describe a process on an unconstrained two-dimensional plane. Additionally, it is also a flexible editor in the sense that it allows process descriptions to exist in a partially specified state and does not force the user to immediately manage all dependencies or to connect all parent-child ports. In all, it provides a flexible mechanism for entering and editing process descriptions.
References


