A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating Software Components

by

Chrysanthos Nicholas Dellarocas

Diploma of Electrical Engineering
National Technical University of Athens, 1989

S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1991

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

Doctor of Philosophy

at the
Massachusetts Institute of Technology
February 1996

Signature of Author _____
Department of Electrical Engineering and Computer Science
December 11, 1995

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

Accepted by _____
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Studies

A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating Software Components

by

Chrysanthos Nicholas Dellarocas

Submitted to the
Department of Electrical Engineering and Computer Science
on December 11, 1995
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

This thesis argues that many of the difficulties associated with building software applications by integrating existing components are related to a failure of current programming languages to recognize component interconnection as a separate design problem, orthogonal to the specification and implementation of a component's core function.

It proposes SYNOPSIS, an architectural description language which supports two orthogonal abstractions: *activities*, for representing the functional pieces of an application, and *dependencies*, for describing their interconnection relationships. *Coordination processes*, defined as an attribute of dependencies, describe implementations of interconnection protocols. Executable systems can be generated from SYNOPSIS descriptions by successively replacing activities with more specialized versions and managing dependencies with coordination processes, until all elements of the description are specific enough for code generation to take place.

Furthermore, it proposes a "design handbook", consisting of a vocabulary of common dependency types and a design space of associated coordination processes. The handbook is based on the insight that many software interconnection relationships can be described using a relatively narrow set of concepts orthogonal to the problem domain of most applications, such as resource flows, resource sharing, and timing dependencies.

A prototype component-based application development tool called SYNTHESIS was developed. SYNTHESIS maintains a repository of increasingly specialized dependency types, based on the proposed design handbook. It assists the generation of executable applications by successive semi-automatic transformations of their SYNOPSIS descriptions.

A set of four experiments is described. Each experiment consisted in specifying a test application as a SYNOPSIS diagram, associating application activities with components exhibiting various mismatches, and using SYNTHESIS to assemble these components into executable systems. SYNTHESIS was able to exploit its dependencies repository in order to resolve a wide range of interoperability and architectural mismatches and integrate independently developed components into the test applications, with minimal or no need for additional manually-written code. It was able to reuse a single SYNOPSIS architectural description in order to generate versions of a test application for two different execution environments. Finally, it was able to suggest various alternative architectures for integrating each component set into its corresponding application.

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

Acknowledgments

This thesis has been the focus of the last three and a half years of my life. As I am getting near the end, I would like to say thanks to all the people and places that have been around me during this time and, each in their own big or small way, have contributed to making this work possible.

Completing a thesis is, first and foremost, an academic exercise. It would not have been possible without a kind and supportive academic advisor. Many thanks to Professor Tom Malone for inviting me to join his research group, for giving me all the freedom and resources I needed to select my topic, and for never losing his faith in me (even during periods when I had lost mine). I am grateful to the members of my thesis committee, Professors Hal Abelson and Peter Szolovits, for their constructive comments and suggestions. Bob Halperin, the executive director of the MIT Center for Coordination Science, deserves my thanks for helping create a stimulating research environment, and for signing each and every one of my research assistantship appointments with a smile. Thanks to George Wyner, my fellow doctoral candidate at CCS, for numerous stimulating conversations (and for being such a nice fellow). Thanks to John Quimby for bringing to my attention the capabilities of the Visio program, one of the crucial components of my prototype implementation. A big hug to Meg Christian, our secretary, for bringing that extra personal touch to the place, and for her very special interest and support (not to mention her wonderful decoration of my office door). Thanks to the ladies of the LCS Reading Room, for creating a peaceful and resourceful environment where most of my literature search took place, and many of my ideas took form. Finally, my warmest thanks to my family for teaching me the value of a good education and for always encouraging my academic pursuits.

Completing a thesis is a balancing act. In my case, I was fortunate to be surrounded by numerous good friends who helped me maintain a (relative) balance during my Ph.D. years and make it through with (relative) sanity. Many thanks to Peter Kofinas, my roommate for the last three years, for being the best approximation to the ideal roommate I have ever met. Thanks to the faithful members of the mailing list `flex@mit.edu` *for motivating me to stay in shape (and for listening to my nonsense during our workouts). Thanks to my close friends Stelios Smirnakis and Michalis Bletsas for always being there for me when I needed them. Besos fuertes to Marina Cocconi for putting up with my doctoral disorders and for being the perfect travel companion to a number of wild and exotic places. A warm kiss to Heleni Vastardis for staying close despite my moods and for being my inseparable Merengue dance partner on many unforgettable occasions.*

Finally, completing a thesis is only one part of a person's life and interests. I was lucky to spend my graduate years at MIT and take advantage of its inexhaustible resources and opportunities for learning. I am grateful to MIT for being such a dynamic, open-minded

place that was able to satisfy all my thirst for knowledge, without ever making me to go to that other school in Cambridge to take a class it did not offer. Special thanks must go to the Sloan School of Management, for allowing me to complement my technical education, and to the Spanish Language Section for enabling me to fulfill uno de los sueños de mi vida.

Last, but not least, thanks to Boston and its wonderful international student community for treating me so well during the five and a half years I have spent here. As I am preparing to move on to the next stage of my life, I am sure I'll miss you all.

Table of Contents

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down as far as I could
To where it bent in the undergrowth;

Robert Frost - *The Road Not Taken*

Chapter 1

Introduction

1.1 Motivation

Many would argue that future breakthroughs in software productivity will depend on our ability to combine existing pieces of software to produce new applications. Less than 15% of the software developed is innovative (i.e. "unique, novel, and specific to individual applications") [Jones84]. The large bulk of the code in our software systems implements solutions to routine design problems, such as sorting, searching, and manipulating well-known data structures.

In mature engineering disciplines, such as chemical or civil engineering, routine design problems are rapidly solved by reusing existing designs [Kogut94a]. The knowledge for routine design is captured in standardized descriptions, organized in design handbooks, and shared within the engineering community. Thus, designers are able to rapidly construct large, high-quality systems, out of well-tested, existing parts.

The potential for reuse certainly exists in the software development domain. In fact, over the past decade, there has been a very significant amount of work in the area of software reuse (see [Biggerstaff89, Krueger92]). However, despite all this effort, software production is still dominated by build-from-scratch techniques.

This work is broadly motivated by the question: *Why is it so hard to build software applications out of existing components?*

One difficulty lies in *locating* the desired components. In contrast to more mature disciplines, software engineering is still lacking comprehensive component libraries and

design handbooks. The convergence of the industry to a small number of dominant designs, plus the easy access to software repositories offered by technologies such as the Internet [Obraczka93], is already making the process of finding appropriate components relatively easy.

But even when the components are at hand, composing them into new applications often requires so much additional effort, that in many cases designers still decide to build their applications from scratch.

Problems arise because the chosen parts "do not fit together well". Designers typically have to either modify the code of existing components, or write additional *coordination software* that bridges mismatches among components. Examples of such mismatches include:

- low-level *interoperability mismatches*, such as differences in expected and provided procedure names, parameter orderings, data types, and calling conventions.

- more fundamental *architectural mismatches*, that is, different assumptions about the structure of the application in which components are to appear. Such mismatches include differences in expected and provided communication protocols, different assumptions about resource ownership and sharing, different assumptions about the presence or absence of particular operating system and hardware capabilities, etc.

Most research efforts to facilitate the process of bridging component mismatches have focused on limited classes of interoperability mismatches, such as interface mismatches [Wileden91, Beach92, Purtilo94] or data type mismatches [Lamb87]. The importance of architectural mismatches has only recently been highlighted [Garlan95]. To this date there has been no unified framework for describing the various kinds of component mismatches, nor a systematic set of rules for dealing with them. Designers still have to rely on their intuition and experience, and the problem of component composition is still being confronted in a largely ad-hoc fashion.

1.2 Research Overview

1.2.1 Objectives

The practical objectives of this work is:

- to better understand why it is currently difficult to build software applications by composing existing components, and

- to propose a novel software system representation and a design framework that aim to reduce the effort of integrating software components into new applications.

1.2.2 Theoretical Argument

This work adopts an architectural perspective on software applications, viewing them as interdependent collections of software components At the architectural level, components and their interdependencies are represented as two distinct, equally important entities. Software components represent the core functional pieces of an application and deal with concepts specific to the application domain. Interdependencies relate to concepts *orthogonal* to the problem domain of most applications, such as transportation and sharing of resources, and synchronization constraints among components.

However, as design moves closer to implementation, current programming tools increasingly focus on representing components. At the implementation level, software systems are sets of source and executable modules in one or more programming languages. Although modules come under a variety of names and flavors (procedures, packages, objects, clusters, etc.) they are all essentially abstractions for components.

By failing to provide separate abstractions for specifying and implementing interconnection protocols among software components, current programming languages force programmers to distribute such protocols among the interdependent components. As a consequence, code-level components encode (apart from their ostensible function) fragments of interconnection protocols from their original development environments. These fragments translate into a set of undocumented assumptions about their dependencies with the rest of the system. When attempting to reuse components in new applications, such assumptions have to be manually identified and modified, in order to match the new interdependency patterns at the target environment. This often requires extensive modifications of existing code, or the development of additional coordination software.

This thesis argues that many of the practical difficulties associated with the above process are due to our current failure to recognize the problem of component interconnection as a separate design problem, *orthogonal* to the problem of implementing a component's core functionality. We need to develop programming languages and tools with support for abstractions that localize and separate the definition of interconnection relationships from that of the interdependent components. Furthermore, we need to understand the patterns of component interdependencies encountered in software systems, and develop frameworks that guide designers in selecting appropriate coordination processes for managing them.

If we can satisfy these requirements, we can develop improved processes for component-based application development. Such processes will center around architectural descriptions of applications, where software activities and their interdependency patterns will be explicitly represented by distinct entities. They will offer the following set of practical benefits:

- *Reuse of code-level components.* Designers will be able to generate new applications simply by selecting existing components to implement activities, and coordination processes to manage dependencies, independently of one another. The development of successful frameworks for mapping dependencies to coordination processes will reduce the step of managing dependencies to a routine one, and enable it to be assisted, or even automated, by design tools. Overall, the objective is to be able to generate new applications with minimal, or no need for user-written coordination software.

- *Reuse of software architectures.* Designers will be able to reuse the same architectural description, in order to reconstruct applications after one or more activities have been replaced by alternative implementations. They will simply have to semi-automatically re-manage the dependencies of the affected activities with the rest of the system. Furthermore, they will be able to reuse the same set of components in different execution environments by managing the dependencies of the *same* architectural description using different coordination processes, appropriate for each environment.

- *Insight into software organization alternatives.* The description of software applications as sets of components interconnected through abstract dependencies will help designers structure their thinking about how to best integrate the components together. A design space of coordination processes for managing interdependency patterns will assist them to explore alternative ways of organizing the same set of components, in order to select the one which exhibits optimal design properties.

1.2.3 Deliverables

The main body of the thesis proposes a concrete implementation of the previous requirements and demonstrates how they can be integrated into a methodology for component-based application development. The following are the principal deliverables of this research:

SYNOPSIS: A software architecture description language

SYNOPSIS supports graphical descriptions of software application architectures at both the specification and the implementation level. It provides separate language entities for

representing software *activities* and *dependencies*. Activities represent the main functional pieces of an application, while dependencies describe their interconnection relationships. An important attribute of dependencies is their *coordination process*. Coordination processes represent interconnection protocols that *manage* the relationships and constraints specified by their associated dependency. SYNOPSIS language elements are connected together through ports. *Ports* provide a general mechanism for representing abstract component interfaces. All elements of the language can contain an arbitrary number of attributes. *Attributes* encode additional properties of the element, as well as compatibility criteria that constrain its connection to other elements.

SYNOPSIS provides two mechanisms for abstraction: *Decomposition* allows new entities to be defined as patterns of simpler ones. It enables the naming, storage, and reuse of designs at the architectural level. *Specialization* allows new entities to be defined as variations of other existing entities. Specialized entities inherit the decomposition and attributes of their parents and can differentiate themselves by modifying any of those elements. Specialization enables the incremental generation of new designs from existing ones, as well as the organization of related designs in concise hierarchies. Finally, it enables the representation of reusable software architectures at various levels of abstraction (from very generic to very specific).

A design handbook of software component interconnection

To assist the design task of specifying application interdependencies, as well as the design of corresponding coordination processes, this work proposes a standardized, but extensible, vocabulary of dependency types and a design space of associated coordination processes. The vocabulary is based on the observation that most software interconnection relationships can be specified using a relatively narrow set of concepts *orthogonal* to the problem domain of most applications, such as resource flows, resource sharing, and timing dependencies. The implementation of interconnection protocols involves a similarly orthogonal set of coordination concepts, such as shared events, invocation mechanisms, and communication protocols. The development of an application-independent framework that captures the most useful patterns of interdependencies and the ways of managing them, can form the basis for a *design handbook for integrating software components*. The development of such a handbook aims to reduce the specification and implementation of software component interdependencies to a routine design problem, capable of being assisted, or even automated, by computer tools.

SYNTHESIS: A component-based application development environment

SYNTHESIS provides an integrated environment for developing software applications by combining existing components. The system provides support for:

- Creating and editing software architectural diagrams written in the SYNOPSIS language.

- Maintaining repositories of SYNOPSIS entities (activities, dependencies, coordination processes, ports), organized as specialization hierarchies.

- Assisting, and in some cases automating, the process of generating executable applications by successive transformations of SYNOPSIS architectural diagrams.

The prototype implementation of SYNTHESIS that has been developed for this thesis contains a version of our component interconnection handbook encoded as a SYNOPSIS specialization hierarchy of dependency types. Using this repository, the SYNTHESIS design assistant is able to semi-automate the process of generating applications from their SYNOPSIS diagrams, by managing dependencies with coordination processes stored in the repository. This results in minimal, and often no need for user-written coordination software to "glue" the components together.

1.2.4 Validation

In order to demonstrate the thesis brought forth in Section 1.2.2, we need to provide positive evidence about:

- the *feasibility* of describing applications as collections of orthogonal subcomponents, representing core activities and interdependencies

- the *usefulness* of such a representation in facilitating both code-level and architectural software reuse, as well as in providing insight into software component organization alternatives.

The feasibility and usefulness of our claims has been demonstrated by building a prototype implementation of SYNTHESIS, and using it to perform a set of four experiments. Each experiment consisted in:

- describing a test application as a SYNOPSIS diagram

- selecting a set of components exhibiting various mismatches to implement activities

- using SYNTHESIS and its repository of dependencies in order to integrate the selected components into an executable system

- exploring alternative executable implementations based on the same set of components

The test applications include:

- a File Viewer system, which integrates heterogeneous components written in C and Visual Basic.

- a Key Word In Context index system, built by assembling components with mismatching architectural assumptions (UNIX filters and servers).

- an Interactive TEX system, which integrates the components of the TEX document typesetting system in a WYSIWYG (what-you-see-is-what-you-get) ensemble.

- a Collaborative Editor architecture, which extends the functionality of existing single user editors with group editing capabilities.

This experience has demonstrated that that the proposed architectural ontology and vocabulary of dependencies were capable of accurately and completely expressing the architecture of all four test applications. Furthermore, it has provided positive evidence for the principal practical claims of the approach. The evidence can be summarized as follows:

- *Support for code-level software reuse:* SYNTHESIS was able to resolve a wide range of interoperability and architectural mismatches and successfully integrate independently developed components into all four test applications, with minimal or no need for user-written coordination software.

- *Support for reuse of software architectures:* SYNTHESIS was able to reuse a configuration-independent SYNOPSIS description of a collaborative editor and the source code of an existing single user editor, in order to generate collaborative editor executables for two different execution environments (UNIX and Windows).

- *Insight into alternative software architectures*: SYNTHESIS was able to suggest a variety of alternative overall architectures for integrating each test set of code-level components into its corresponding application, thus helping designers explore alternative designs.

1.3 Related Work

Both the problem of representing software application architectures, and that of developing applications from existing components have received attention in a variety of

different research areas. This section briefly discusses the three research areas that are most closely related to this work. Chapter 7 is devoted to a detailed discussion of other related research.

Coordination Theory

Coordination theory [Malone94] focuses on the interdisciplinary study of coordination. Research in this area uses and extends ideas about coordination from disciplines such as computer science, organization theory, operations research, economics, linguistics, and psychology. It defines coordination as the process of managing dependencies among activities. Its research agenda includes characterizing different kinds of dependencies and identifying the coordination processes that can be used to manage them.

This work is directly related to coordination theory, in that it views the process of developing applications as one of specifying architectures in which patterns of dependencies among software activities are eventually managed by coordination processes.

It makes two principal contributions to coordination theory:

- The development of SYNOPSIS, an architectural language that is able to support coordination theory research, with distinct abstractions for activities and dependencies. Although developed for describing software applications, SYNOPSIS can be used to describe other complex systems as well, such as business processes.

- The definition of a vocabulary of dependency types and associated coordination processes for the domain of software systems.

This project grew out of the Process Handbook project [Malone93, Dellarocas94] which applies the ideas of coordination theory to the representation and design of business processes. The goal of the Process Handbook project is to provide a firmer theoretical and empirical foundation for such tasks as enterprise modeling, enterprise integration, and process re-engineering. The project includes (1) collecting examples of how different organizations perform similar processes, and (2) representing these examples in an on-line "Process Handbook" which includes the relative advantages of the alternatives.

The Process Handbook relies on a representation of business processes that distinguishes between activities and dependencies and supports entity specialization. It builds repositories of alternative ways of performing specific business functions, represented at various levels of abstraction. SYNOPSIS has borrowed the ideas of separating activities from dependencies and the notion of entity specialization from the Process Handbook. It is especially concerned with (1) refining the process representation so that it can describe software applications at a level precise enough for code generation to take place, and (2)

populating repositories of dependencies and coordination processes for the specialized domain of software component integration.

Architecture Description Languages

Architecture Description Languages (ADLs) provide support for representing software systems in terms of their components and their interconnections [Kogut94b, Shaw94b]. They are used to model domain-specific software architectures so that new application systems can be built from existing solutions. They typically provide separate abstractions for representing components and their interconnections.

SYNOPSIS shares many of the goals and principles of ADLs. However, whereas previously proposed architectural languages only provide support for implementation-level connector abstractions (such as a pipe, or a client/server protocol), SYNOPSIS is the first language which also supports specification-level abstractions for encoding interconnection relationships (dependencies). It is also the first system that proposes a design framework for describing the most common interconnection relationships encountered in software systems, and for associating each relationship to appropriate implementations (coordination processes).

Operating, Concurrent, and Distributed System Design

Operating system research is concerned with developing algorithms for the allocation of system resources and the management of interdependencies among user and system processes. The study of concurrent and distributed systems in concerned with essentially the same problems in the special domains of systems with multiple processors or physically distributed components [Andrews91, Bacon93].

This work provides a unifying framework for organizing the techniques and algorithms developed in those areas, relates them to dependency patterns, and uses them to populate the design space of coordination processes for software systems.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows:

Chapter 2 contains a detailed exposition of the underlying thesis of this work. It explains why current technologies for reusing software components into new applications often require significant additional design effort to resolve component mismatches. It argues for treating component interconnection as a separate design problem, orthogonal to the

problem of implementing a component's core function. It outlines a set of practical requirements to achieve this separation and a set of practical benefits that are expected to result from it.

Chapter 3 is devoted to a description of SYNOPSIS, a software architecture description language. Its main distinguishing feature is a clear separation of an application's core functional pieces from their interconnection relationships.

Chapter 4 introduces a vocabulary of dependencies for describing software component interconnection needs. It also describes a design space of associated coordination processes. The vocabulary and design space can form the basis of a design handbook of software component interconnection, that facilitates the representation and solution of component interconnection problems.

Chapter 5 describes an algorithm for generating executable applications by successive transformations of their SYNOPSIS architectures. The algorithm forms the basis of the SYNTHESIS design assistant.

Chapter 6 describes the prototype implementation of SYNTHESIS. It also contains a detailed discussion of four experiments which demonstrate the feasibility and usefulness of the ideas proposed in this thesis.

Chapter 7 contains a detailed discussion of related research.

Finally, Chapter 8 summarizes the results of this work, distills some lessons learned, and discusses some ideas for future research.

Chapter 2

Component Interdependencies Deserve First-Class Status

In this chapter we seek to understand why current technologies for composing software components into new applications often require so much additional design effort, as to make the development of applications from scratch preferable to the reuse of existing components. We analyze the process of component composition and argue that the difficulties of performing its steps are closely related to the failure of current programming languages and design tools to separate the implementation of component interconnection protocols from that of the "core" function of each component. We then argue that the composition effort can be reduced, if we separate the design problem of component interconnection from that of implementing a component's core function, and begin to comprehend, represent and systematize its design dimensions and design alternatives in a coherent framework. This argument forms the principal thesis of the work. We outline a set of notations, theories and tools needed to support this separation. We propose a new, improved component composition process based on them. This sets the stage for the rest of the thesis, which presents one concrete implementation of those requirements and demonstrates how they can form the basis of a useful methodology for component-based application development and maintenance.

2.1 Software Applications are Interdependent Sets of Activities

When a software engineer begins the design of a new software application, he/she often draws a diagram labeled the "software architecture". The diagram usually consists of boxes, depicting the main functional pieces of the application, and lines, depicting

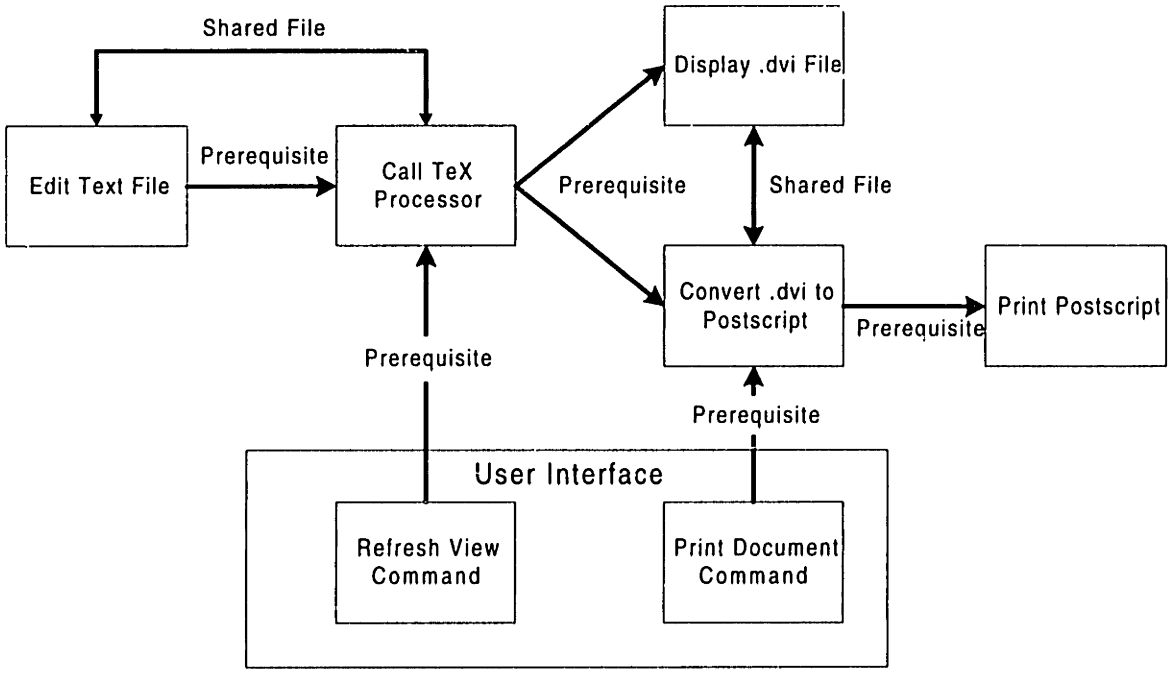


Figure 2-1: A software architecture for a document processing application. Boxes correspond to activities while labeled arcs represent dependencies.

interconnection relationships among functional pieces. For example, Figure 2-1 shows one way of representing the architecture of a document processing application based on the TEX system.

At this level of description, software applications are systems of interdependent software activities. In this thesis, we will use the term *activities* to describe the major functional elements of an application. We will use the term *dependencies* to describe relationships and constraints among activities, such as data and control flows, timing constraints, or resource sharing relationships. Table 2-1 lists some dependency types commonly encountered in software systems.

Software architecture diagrams are *specification-level* descriptions of software systems. In order to obtain the desired behavior of an application, activities must be implemented by *executable activities* or *software components*[1] and dependencies must be managed by appropriate *coordination processes*. Software components are software entities of various forms, including source code modules, executable programs, or remote network services. Coordination processes implement component interconnection protocols, such as pipe

[1] The two terms will be used interchangeably in the rest of the thesis.

Dependency Type	*Design Dimensions*	*Coordination Processes*
Prerequisite	- number of precedents - number of consequents - relationship among precedents (And/Or)	- Explicit Notification - Event Polling
Mutual Exclusion	- number of participants	- Semaphore protocol - Token Ring protocol
Data Flow	- number of producers - number of consumers - data type(s) produced - data type(s) consumed	- Shared Memory protocol - Pipe protocol - Client/Server protocol
Shared Resource	- number of users - modes of usage - resource sharing properties	- Time-Sharing - Paging - Replication

Table 2-1: Some common types of dependencies encountered in software systems and representative coordination processes for managing them.

channels (manage one-to-one flow dependencies), client/server organizations (manage many-to-one flow dependencies), semaphore protocols (manage mutual exclusion dependencies), etc.

Some coordination processes correspond to atomic language or operating system protocols (such as a simple procedure call). Other describe composite protocols which introduce additional activities and dependencies to be integrated into the rest of the system. It is important to observe that, whereas software activities relate to concepts specific to the application domain, interdependency patterns relate to concepts *orthogonal* to the problem domain of most applications, such as transportation and sharing of resources, or synchronization constraints among components.

Example 2-1: Dependencies and Coordination Processes

Figure 2-2 depicts a simple architecture where a one-to-one flow dependency connects two activities. One way of managing this dependency is by setting up a pipe protocol between the two activities. The pipe protocol introduces three new activities and two new dependencies into the system.

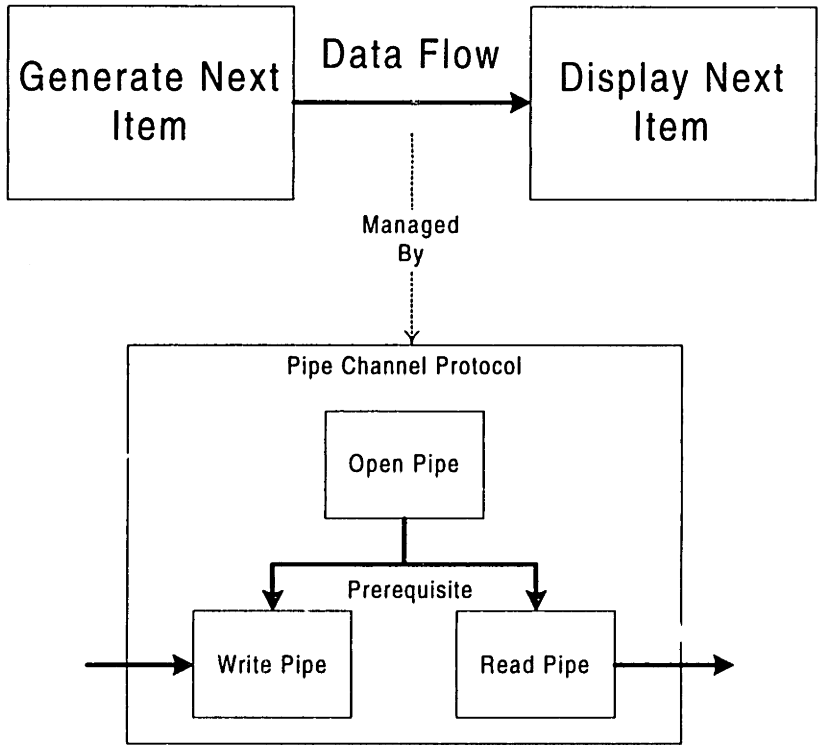


Figure 2-2: A simple application architecture and a coordination process for managing a data flow dependency between its activities.

Example 2-2: Dependency patterns define system behavior

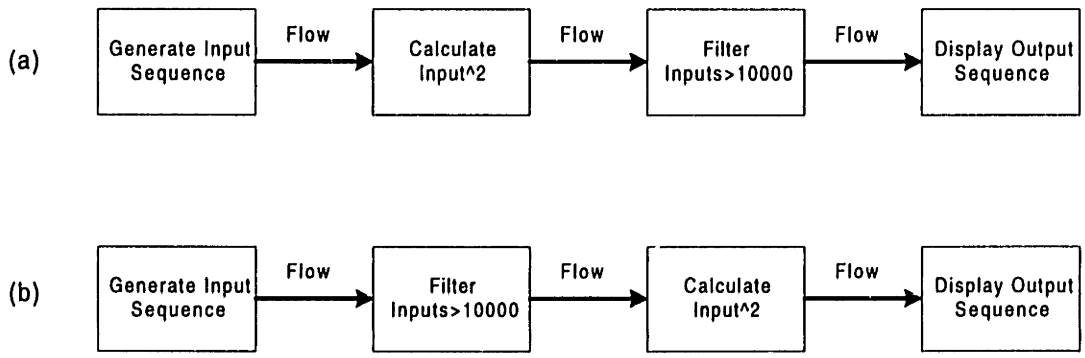


Figure 2-3: Interconnecting the same activities in two different ways results in two different applications.

The purpose of this example is to demonstrate the equal importance of activities and dependencies in determining the behavior of an application. Figure 2-3 shows the architecture of two applications where an identical set of activities has been interconnected in two different ways. The behavior of the two applications is quite different. Application (a) outputs the squares of all members of its input sequence that are below 100 (since $100^2 = 10000$). Application (b) outputs the squares of all members of its input sequence that are below 10000.

In conclusion, at the architectural level activities and dependencies are two distinct elements of complex software applications. Both are explicitly represented and both are equally important to the definition of a system.

2.2 Implementation Languages Focus on Components

Despite the equal status of activities and interdependencies in informal descriptions of a software architecture, as design moves closer to implementation, current design and programming tools increasingly focus on components, leaving the description of interconnections among components implicit, distributed, and often difficult to identify. At the implementation level, software systems are sets of source and executable modules in one or more programming languages. Although modules come under a variety of names and flavors (procedures, packages, objects, clusters etc.), they are all essentially abstractions for components.

Traditional language mechanisms for specifying module interconnections are essentially variations of the procedure call interface. They specify input and output parameters, and in some cases, lists of imported and exported procedures. They are not sufficient for describing more complex interconnections that are commonplace in today's software systems (for example, a distributed mutual exclusion protocol). As a consequence, support for complex module interconnections is either left implicit, relying on the semantics of programming languages and operating systems, or is broken down into fragments, and embedded within the modules themselves.

Example 2-3: Distribution of Component Interconnection Protocols

This example will demonstrate how protocols for implementing even the simplest interconnection relationships, become fragmented and distributed among several code modules when the system is implemented in a conventional programming language.

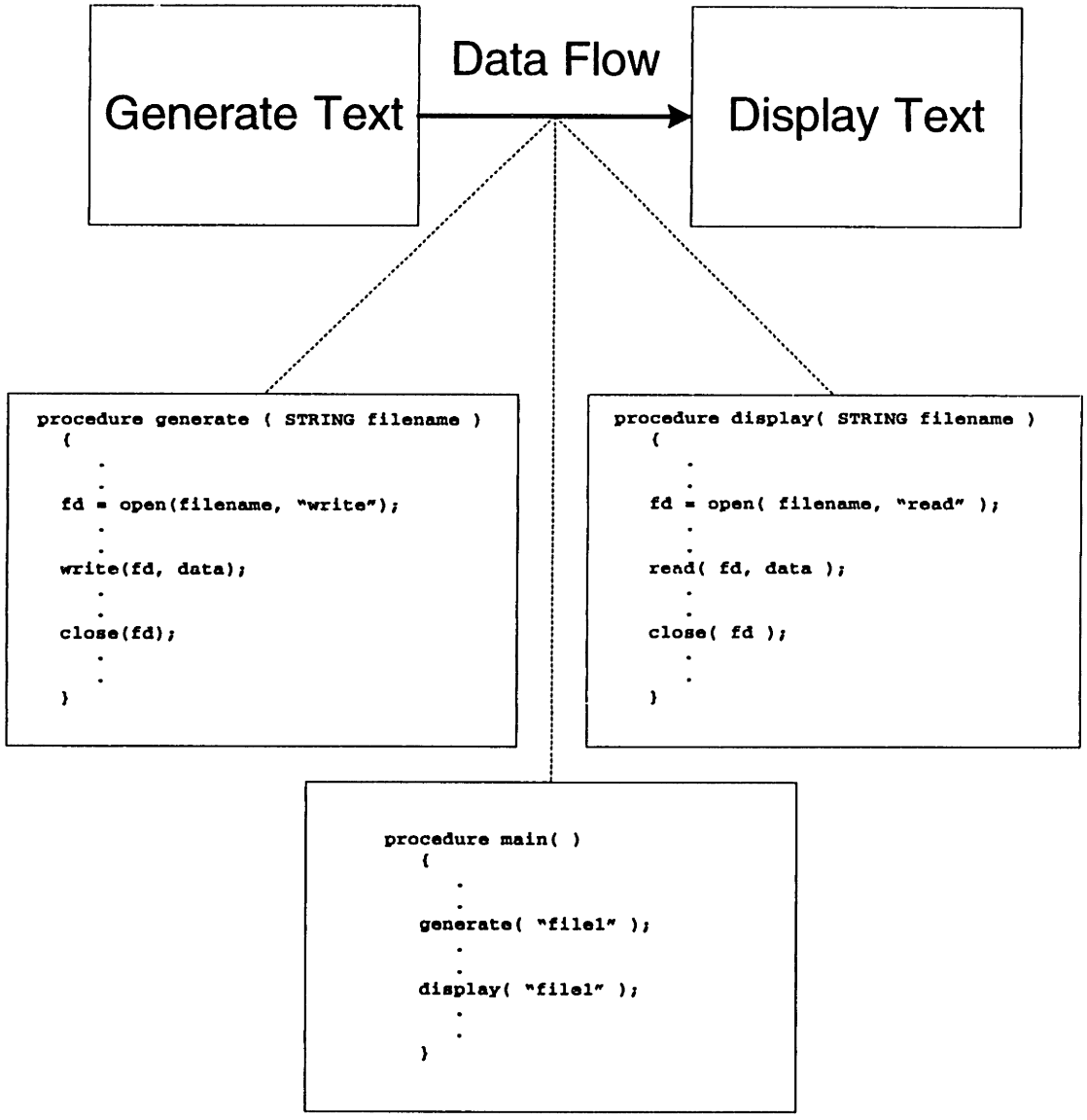


Figure 2-4: Implementation languages distribute support for component interconnection protocols among several code modules.

Figure 2-4 shows the architecture and one implementation of a simple application that generates, and then displays, a body of text. Management of the single flow dependency requires a mechanism for transporting the text from activity Generate Text to activity Display Text. It also requires a mechanism for ensuring that activity Display Text reads the text only after activity Generate Text has written it. In this particular implementation we have chosen to manage the flow dependency among the two application activities using a sequential file. Component `generate` opens, writes and closes the file, while component `display` opens, reads, and closes the file. The main program passes the shared filename to both components.

In this program, the management of a single data flow dependency has been distributed to three different places. Procedures `generate` and `display` each contain one part of the file transfer protocol embedded in their code body. Less obviously, the management of the read-after-write prerequisite assumption is being implicitly managed by ordering the invocation of `generate` before that of `display` in the (sequential) main program.

The distribution of even such a simple protocol in three different modules has the consequence that each of the resulting modules now contains certain *implicit* interconnection assumptions about other modules with which it will work together. For example, module `display` assumes that, by the time it begins execution, some other module will have written the data it needs to a file. When attempting to reuse `display` in a different context, these assumptions will quite often not hold. In those cases, the code of the module must be manually modified to match the interconnection assumptions that are valid in the new context.

A visual metaphor that might be useful in order to understand the loss in expressive power as we move from architectural diagrams to system implementations, is the separation of a 3-dimensional image into its projections on a set of 2-dimensional planes. Let us try to picture a software application as a set of black boxes, each corresponding to a functional component. In 3-dimensional space, black boxes are interconnected by 3-dimensional connectors, which manage their interdependencies. This corresponds to the architectural view of an application. Using this metaphor, the lack of explicit support for representation of interconnections at the implementation level corresponds to an inability to represent the third dimension. Programs are thus like sets of 2-dimensional planes. In this metaphor, planes correspond to code-level modules. In order to map a 3-dimensional image into a set of planes, we take its projection into each of the planes. At the end of the process, each plane encodes a "core" functional component but also contains the trace of an interconnection protocol embedded in it (Figure 2-5).

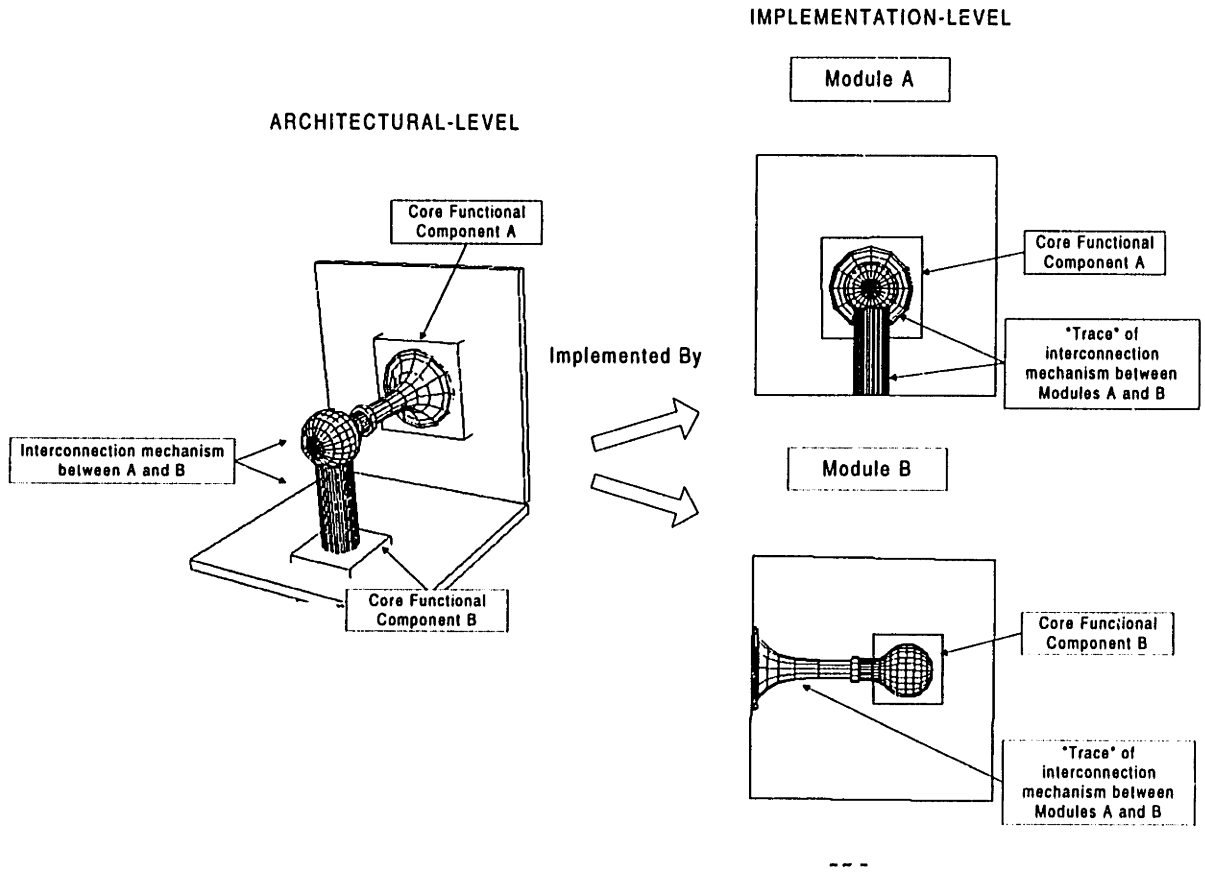


Figure 2-5: Moving from architectural diagrams to system implementation is similar to projecting a 3-d image onto a set of 2-d planes.

Trying to understand a system by reading its code modules is equivalent to trying to mentally reconstruct a 3-d image from its 2-d projections. The separation of interconnections into a set of 2-dimensional traces, blurs the ostensible function of each module and makes global understanding of the interdependency patterns a nontrivial mental exercise. Furthermore, as we shall see, it has negative implications on the reusability of modules thus constructed.

Several researchers [Shaw94a, Allen94] have identified this expressive shortcoming of programming languages and systems and have discussed its negative effects in developing and maintaining software systems. This thesis makes the additional argument that the lack of explicit support for representing component interdependencies and implementing component interconnections separately from the components themselves, is one of the principal causes for the difficulty of reusing software components in new applications. The following sections will attempt to make this argument more explicit.

2.3 Implicit Interconnection Assumptions Pose Obstacles to Component Reuse

Implementation-level software modules (source or executable) are the usual candidate units for software reuse. This section will discuss why the current practice of ignoring the significance of architectural interdependencies and embedding component interconnections within components, increases the difficulty of reusing these components in new applications.

As described in the previous section, components built with current technologies make strong, and usually undocumented, assumptions about their interdependencies and interconnections with other components in the same application. More precisely, they contain fragments of protocols that implement their interconnections with the rest of the system in their original development environment. Such assumptions are either hardwired into their interfaces, embedded in their code bodies, or left implicit, relying or specific properties of the environment for which they were originally designed.

Since interconnection relationships is one of the defining elements of each individual application, different applications are expected to contain different interconnection relationships. Therefore, the interaction assumptions embedded inside a component will most likely not match its interconnection patterns in the target application. In order to ensure interoperability, the original assumptions have to be identified, and subsequently replaced or bridged with the valid assumptions for the target application. In many cases this requires extensive code modifications or the writing of additional *coordination software* (or *"glue" code*).

2.3.1 A Taxonomy of Interconnection Assumptions

Since the problem of component composition is closely related to that of managing component interconnection assumptions, it is useful at this point to introduce a framework that characterizes the various categories of interconnection assumptions encountered in reusable software components. Our framework classifies assumptions according to the following two dimensions:

a. The *design level* (specification, implementation) of the assumption
b. The *location* of the assumption in the component

With respect to their design level, interconnection assumptions are classified into:

- *Protocol assumptions.* These are implementation-level assumptions which result from the fact that components encode parts of their interconnection protocols into their bodies. For example, a producer module which was originally designed as a UNIX

filter, contains the writer part of the pipe protocol and assumes it will be interacting with modules which encode the corresponding reader part of the protocol.

- *Architectural assumptions.* These are specification-level equivalents of protocol assumptions. They can be expressed as constraints on the patterns of allowed interconnection relationships between a component and the rest of the system. Take, for example, the previously mentioned producer module, implemented as a UNIX filter. Since pipe protocols manage one-to-one flow dependencies (each value written to a pipe can only be read once), that module contains the architectural assumption that each of its values will flow to a single consumer component. Architectural assumptions are more difficult to identify than protocol assumptions, because there exist no systematic frameworks for mapping interconnection relationships to interconnection protocols, and vice-versa.

With respect to their location inside a component, interconnection assumptions are classified into:

- *Interface assumptions.* These are protocol and architectural assumptions encoded into a module's parameter and import/export interface. Let's consider a simple procedure call as an example. Protocol assumptions are the most obvious: they include the names, data types, and ordering of parameters it expects. However, procedure interfaces also encode a set of implicit, architectural assumptions: These include the fact that all input parameters are expected to flow from a single place (the point of call), that all parameters will be available when control flows into the procedure body, and that output parameters flow to the same place that inputs came from.

- *Assumptions embedded in the code.* These assumptions relate to interconnection protocol fragments found in the code bodies of components and their associated dependency patterns. For example, a Dynamic Data Exchange (DDE) server module written for the Microsoft Windows environment, contains the server part of a DDE protocol embedded in its code. It assumes it will be interacting with DDE client components containing the corresponding client part of the protocol. From an architectural perspective, it assumes it will be connected to other components through many-to-one flow dependencies.

- *Implicit assumptions.* These assumptions are manifested by the *absence* of any particular mention in code and rely instead on properties of the environment for which components were originally designed. An example is a user interface event loop, originally designed for an environment with preemptive scheduling. It implicitly assumes it will be sharing the processor with all other applications running in the system. Therefore, it does not contain statements that periodically yield the processor. The explicit addition of such statements would be required in environments without

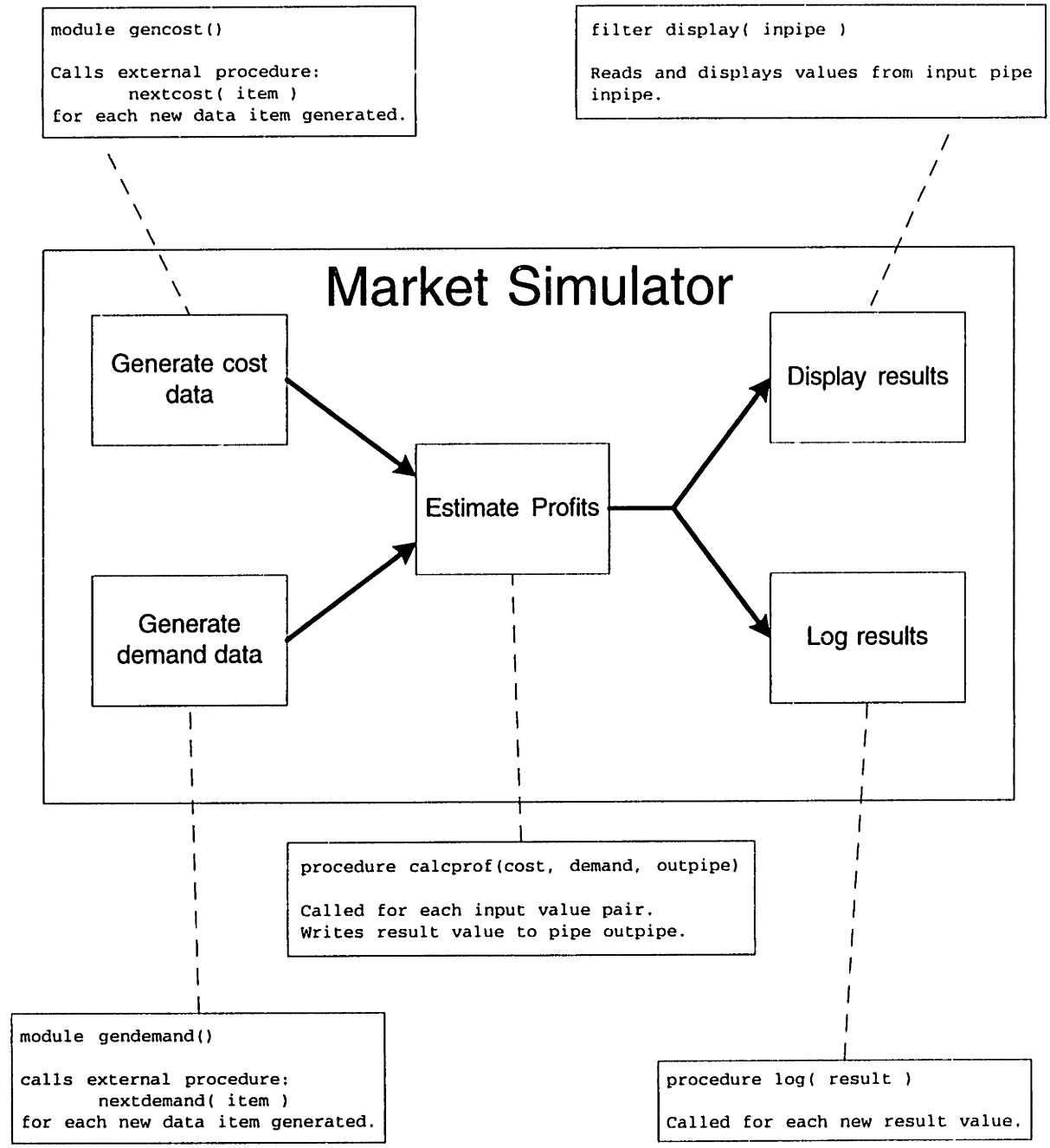


Figure 2-6: Architecture of a simple application and descriptions of candidate components for implementing its activities.

preemptive scheduling. Another example is a module which assumes it will be the only writer of a file in the system and thus does not contain any statements for locking the file. In order to use the module in an environment with multiple concurrent file writers, a file locking protocol must be introduced.

Example 2-4: Construction of a simple application from existing components

The following example demonstrates how the difficulties of reusing software components in new applications are a consequence of mismatches between the original component interconnection assumptions and the actual interdependency patterns of the target application.

Figure 2-6 shows the architecture of a simple market simulator application. We wish to build the application using existing components, whose interfaces and requirements are also summarized in the same diagram.

The first important mismatch occurs at the inputs of activity Estimate Profits. Procedure `calcprof` expects to receive both its input parameters, via a procedure call (protocol assumption), from a single place (architectural assumption). However, in this application, it is connected to two different activities, each of which passes its respective parameter to a different procedure. Module `gencost` repeatedly calls procedure `nextcost` for each new cost data item it generates. Likewise, module `gendemand` independently calls procedure `nextdemand` for each new demand data item it generates. Somehow the two values must be collected together and transmitted, via a single procedure call, to `calcprof`. One way to achieve this is to write one of the values to a global variable, and then read that variable and call `calcprof` from inside the procedure to which the other value has been passed. Such a scheme requires additional coordination to ensure that (a) the global variable is not read by the second procedure before it has been written by the first procedure, and (b) that the first procedure does not overwrite the variable before the second procedure has read and transmitted each value.

The other mismatch occurs at the output of activity Estimate Profits. Since `calcprof` writes its output values to a pipe, it implicitly assumes (a) that there will be a pipe reader at the other end (protocol assumption), and (b) that each value will be used by a single consumer activity (architectural assumption). In this example, however, two activities need to read each value. In order for this to happen, additional software must be provided that reads the pipe and makes each value available to both consumer activities. In the case of activity Display results this involves setting up a second pipe. In the case of activity Log results a simple procedure call is sufficient.

Step 1:	**Identify original component assumptions** Identify the interconnection assumptions of each component.
Step 2:	**Determine new patterns of interaction** Determine the patterns of interconnection of each component with the rest of the system in the target application.
Step 3:	**Mediate component assumption mismatches** Replace or mediate the original component assumptions in order to make them compatible with the patterns of interconnection in the target application.

Figure 2-7: Conventional view of the process for building software applications from existing components.

We observe that, even in a simple system, the process of component composition is not trivial. In more complex systems, carrying out the above process might involve significant design effort, to the extent that many designers prefer to build new applications from scratch rather than reuse existing components they might have at hand. Progress has been made in automating the handling of some of the most "obvious" categories of protocol assumption mismatches, such as interface assumptions. However, up to this date, there has been little success in handling more complex protocol assumptions, as well as in treating all different scurces of assumption mismatches under a unifying framework. Researchers have only recently began to recognize the importance of architectural assumption mismatches [Garlan95].

We claim that an important source of difficulties in this area is due to our failure to recognize component interconnection as a separate design problem, provide tools for visualizing component interdependencies, and separate support for component interactions from the implementation of a component's core function. Most designers ignore (or do not consciously take into account) the dimension of component interdependencies and view the process of component composition as an ad-hoc effort to bridge mismatches among components (see Figure 2-7). This view of the process results in a number of practical difficulties:

- Difficulties in identifying component assumptions
- Difficulties in determining new patterns of interaction
- Difficulties in replacing or mediating component assumptions

The following sections elaborate on each of these difficulties.

2.3.2 Difficulties in Identifying Component Assumptions

By not recognizing component interconnections as an explicit element of software systems, current programming languages leave them implicit and often not documented. Designers have to understand or deduce them by reverse-engineering the code. As explained in Section 2.3.1, relevant assumptions might be hidden in a variety of places, including:

- *Module interfaces.* Such assumptions include the names, order and data type of input and output parameters, as well as names and signatures of imported and exported modules. They are the most obvious and easy to identify assumptions. Researchers have developed special notations, called Module Interconnection Languages (MILs), to facilitate this task (see Section 7.3.1). Unfortunately, they don't tell the whole story.

- *Code blocks.* Apart from procedure argument lists, there is a large variety of other means of component interaction, ranging from global variables, to complex interprocess communication protocols requiring elaborate setup and maintenance. Each participating component will contain fragments of those protocols embedded in its code blocks. Although good program design is able to help localize such fragments, nothing in today's languages actually forces programmers to do so. In many cases, designers have to read the entire code in order to identify them.

- *Implicit environmental properties.* The most problematic assumptions are those which are manifested by the absence of any particular mention in code and rely instead on properties of the environment for which components were originally designed. There is no tangible documentation of such assumptions, except in program comments. In the worst case, designers must rely on their experience, intuition, and knowledge of the development environment of the component.

2.3.3 Difficulties in Determining New Patterns of Interaction

This is the step where the lack of support for explicit representation and management of component interdependencies creates the most problems. The interactions of components

in the target application result from the management of their interdependencies in the new system. Designers need to understand those new patterns of interdependencies, map them to appropriate coordination processes that manage them and, finally, determine the pieces of those processes that are associated with each component in the system. We see, therefore, that this step should, in fact, decompose into three distinct steps, and be supported by explicit representations of the target application architecture and frameworks for mapping interdependency patterns to coordination processes.

An additional complexity comes from the fact that interconnection protocols typically deal with issues orthogonal to those of the problem domain of an application, such as machine architectures, operating system mechanisms and communication protocols. This increases the expertise requirements from the part of the designer and complicates the mental effort needed to mix those orthogonal concerns in the same code unit.

2.3.4 Difficulties in Replacing or Mediating Component Assumptions

Since original component assumptions might be encoded in a variety of places, replacing or mediating them might require several modifications within or around a component. Furthermore, since target coordination processes typically define a set of "roles" to be integrated with several interdependent components, this, again, usually results in the need to perform several modifications distributed across the system.

The previous difficulties can, again, be illustrated by the 3-d-to-2-d-transformation metaphor of the component composition process (see Section 2.2). The input to the process (a set of modules to reuse) corresponds to a set of 2-d planes, each of which encodes a specific component, but also contains the trace of interconnections of this component in its original environment. The output of the process corresponds to another set of 2-d planes (the input set with possibly some new modules added). The process consists in modifying the input planes, erasing or modifying the projection of original interconnection protocols, so that at the end of the process it has become equal to the equivalent projection of the target interconnection protocols.

The visual process is greatly facilitated by the existence of an intermediate 3-d representation, picturing interconnection relationships among components in the target application. Managing application interdependencies with coordination processes and decomposing processes into pieces associated with each component is equivalent to projecting the new 3-d model onto the set of 2-d planes that represent the original modules. It would be very hard to imagine the form of the new projections without recourse to this 3-dimensional view.

The problem with current technologies is that they do not provide support for the equivalent of a 3-d view of software systems, nor a systematic way of projecting "3-dimensional" interconnection protocols onto existing "2-dimensional" software modules.

Even if the inputs and outputs of the process are still modules expressed in our current "2-dimensional" idioms, the existence of an intermediate "3-dimensional" architectural view and algorithms for performing the equivalent of a projection of that view to "2-dimensional" modules, would greatly facilitate the task of component composition.

2.4 Component Interdependencies Deserve First-Class Status

The preceding discussion provides the motivation for the main thesis of this work. We have shown that the complexity of the process of constructing software applications from existing components is a consequence of our failure to separate context-specific support for component interconnections from the implementation of a component's "core" functionality.

We define *software interconnection* to be the design problem of:

(a) specifying the patterns of dependencies among activities in a software application
(b) selecting coordination processes to manage dependencies and integrating the resulting interconnection protocols with the rest of the system

This thesis argues that the problem of component interconnection should be treated as an *orthogonal* design problem from that of the specification and implementation of a component's core function. We should begin to comprehend, represent and systematize its design dimensions and design alternatives in a coherent framework. Such a separation will provide benefits, not only for the initial development, but also for the maintenance and portability of component-based applications.

2.4.1 Requirements for Separating Interconnection from Implementation

The proposed separation translates to a number of concrete requirements for new notations, tools and theories to support the process of component composition. These requirements relate directly to the discussion of the difficulties presented in the Section 2.3, and can be summarized as follows:

2.4.1.1 Make component assumptions explicit

A full separation of the problem of component interconnection from that of component implementation would require us to design components with minimal interdependency assumptions. In actual practice such a requirement would limit the reuse of components developed with current technologies. It might also pose stringent constraints to the development of new components. A more relaxed requirement is to develop component description notations that help make interconnection assumptions explicit.

2.4.1.2 Separate and localize representations of component interdependencies

We need to be able to represent component interdependencies separately from components and to localize interconnection protocols that manage them. This requires the development of architectural languages with separate abstractions for software activities, dependencies, and coordination processes. It also requires the development of a vocabulary for specifying common component interdependencies.

2.4.1.3 Develop systematic design guidelines for component interconnection

We need systematic frameworks for selecting coordination processes that manage application interdependencies. Such frameworks should enumerate the design dimensions of the problem of component interconnection. They should provide mappings from patterns of dependencies to sets of alternative coordination processes for managing them. Finally, they should provide compatibility criteria and design rules for selecting among alternative design choices.

Step 1: **Specify target application architecture**

Express the architecture of the target application as a set of activities interconnected through dependencies.

Step 2: **Associate activities to code-level components**

Locate and select existing code-level components that implement the core functionality of application activities.

Step 3: **Manage target application interdependencies**

Select coordination processes that manage target application interdependencies.

Step 4: **Integrate components and coordination processes**

Integrate code-level components and coordination processes into sets of executable modules.

Figure 2-8: An improved approach for building software applications from existing components.

2.4.2 Benefits from Separating Interconnection from Implementation

If we are able to satisfy the above requirements, we can develop improved processes for developing and maintaining software applications out of existing software components. Such processes will center around explicit, abstract descriptions of software applications at the architectural level. Figure 2-8 gives a generic description of such an approach and Figure 2-9 a schematic high-level diagram of the transformations involved.

Successful implementations of such processes should be able to offer the following practical benefits:

2.4.2.1 Benefits to initial application development

- *Independent selection of components.* Designers should be able to select components to implement activities independently of one another. Candidate software components need not conform to any particular set of standards or assumptions. Any mismatches will be handled by coordination processes.

- *Routine management of dependencies.* Dependencies should be routinely managed by coordination processes based on systematic design frameworks. Depending on how successfully frameworks are developed, the process of dependency management should be assisted, or even be automated, by design tools.

- *Minimal or no need for user-written coordination code.* Technologies for mediating original component assumptions and integrating components and coordination processes into sets of executable modules should be able to generate new applications with minimal, or no need for additional, user-written coordination code.

2.4.2.2 Benefits to application maintenance

- *Easy replacement of components with alternative implementations.* Designers often need to change the code-level components that implement the functionality of specific activities, in order to reflect changes in functional requirements, or to take advantage of new, improved software products. Applications should be easily reconstructed after such changes, by reusing the same architectural diagram and simply managing again the dependencies of the affected activities with the rest of the system.

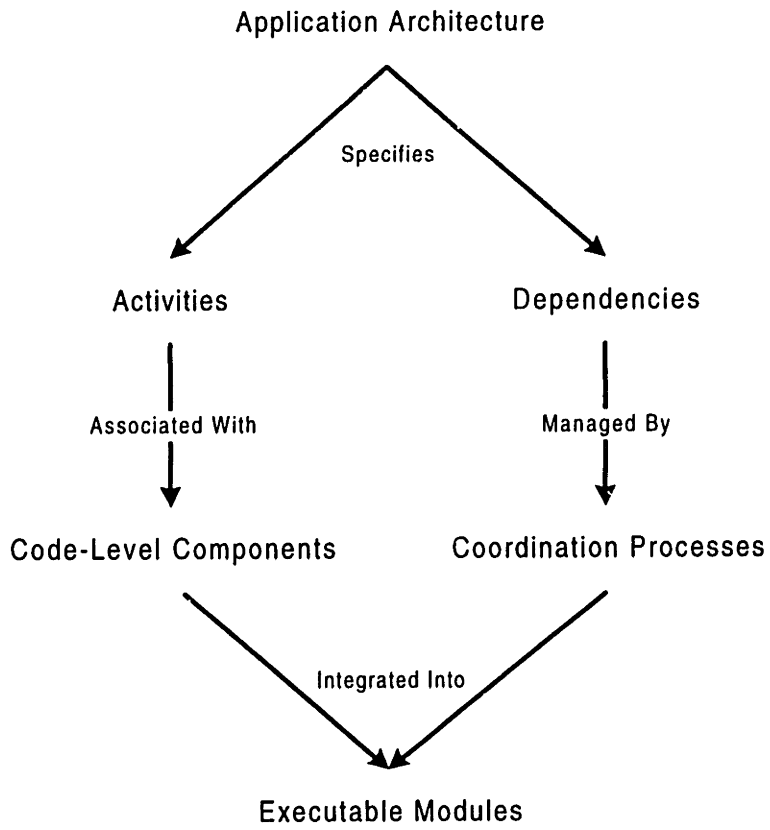


Figure 2-9: A schematic view of the proposed process for building component-based applications.

- *Easy porting of applications to new configurations.* When applications are ported to a new environment, their abstract architecture (activities and dependencies) remains unaffected. However, the use of different coordination processes might be required. By making the step of dependency management a routine one, it should be easy to manage them again for the new environment and construct a new application from the original architectural description and functional components.

2.5 The Way Ahead

Our objective in the rest of this thesis is to propose one concrete implementation of the requirements presented in Section 2.4.1 and to demonstrate how they can be combined into a useful methodology for component-based application development. The

methodology is based on the process sketched in Figure 2-9 and offers the practical benefits outlined in Section 2.4.2. The end products of the work include:

- An architectural language (SYNOPSIS) for describing software applications, with explicit support for activities, dependencies, and coordination processes. Chapter 3 is devoted to a detailed description of SYNOPSIS.

- A vocabulary of commonly encountered activity interdependencies and a design space of associated coordination processes. Chapter 4 presents the vocabulary and design space.

- An application development tool (SYNTHESIS) that allows designers to enter architectural descriptions of applications and is able to assist the process of generating executable systems by successive semi-automatic specializations of their architecture.

Chapter 5 proposes one concrete implementation of the process outlined in Figures 2-8 and 2-9. The design assistant component of the SYNTHESIS system is based on that process. Finally, Chapter 6 presents a number of experiments that test and validate the feasibility and usefulness of the entire approach.

Chapter 3

SYNOPSIS: A Software Architecture Description Language

In this chapter, we begin the description of the principal end products of this research. Our overarching objective, argued for in the previous chapter, is to support a more efficient process for developing software applications out of existing parts. The process generates new applications by applying a series of computer-assisted transformations to descriptions of their architecture. The purpose of this chapter is to describe the first requirement for supporting the process: An architecture description language called SYNOPSIS[1]. The language provides linguistic support for representing and transforming software architecture diagrams. It has two principal distinguishing features: First, it enables a clear separation of an application's core functional pieces from their interdependencies. Second, it supports entity specialization, a mechanism through which software architectures can be described at various levels of abstraction. The chapter begins by outlining the major design objectives of SYNOPSIS. It then gives an overview of the language, based on a simple example. The main body of the chapter is devoted to a detailed discussion of SYNOPSIS language elements, element transformations, and compatibility checking mechanism.

3.1 Language Design Objectives

The purpose of SYNOPSIS is to provide linguistic support for representing and transforming the entities of the component-based application development process that was outlined at the end of the previous chapter (see Figure 2-8). More specifically, support is needed in the following three areas:

---

[1] **syn·op·sis** (si näp'sis) ***n.*** [from Greek *syn-*, together + *opsis*, a seeing, visual image] a statement giving a brief, general review or condensation, summary

3.1.1 Representation of Application Architectures

It should be possible to clearly separate the core functional pieces of an application from their application-specific patterns of interdependencies.

Chapter 2 presents our detailed argument for developing software system description notations with this property. SYNOPSIS satisfies this requirement by supporting two distinct language elements: *activities*, for representing core functional parts and *dependencies*, for representing interconnection relationships among activities.

It should be possible to separate the specification of system architecture from the implementation of the elements being structured.

Such a separation is desirable in order to be able to:

- select or replace components for implementing activities independently of one another (their specification-level interdependencies do not change; different component implementations might influence the implementation of interaction protocols)

- port an application to different configurations (specification-level relationships do not change; implementation of interaction protocols is the element most likely to change across configurations)

SYNOPSIS satisfies this requirement by representing implementation-level entities as optional attributes of activities and dependencies. Thus, activities can optionally be associated with a code-level *component*, such as a source code module, an executable program, or a network server. Dependencies can optionally be associated with a *coordination process*, representing a protocol for managing the relationship described by the dependency.

Section 3.3 describes SYNOPSIS language elements in detail.

3.1.2 Transformation of Application Architectures

It should be possible to connect the specification of system architecture and the implementation of the elements being structured through well-defined, structural transformations.

This requirement is instrumental to allowing the generation of executable applications directly from their architectural diagrams. SYNOPSIS supports the mechanism of entity specialization which allows generic application architectures to be transformed to

executable systems by successive specialization of activities and dependencies contained in their decomposition.

Section 3.5 describes the mechanism of entity specialization.

3.1.3 Support for Compatibility Checking

It should be possible to specify and check compatibility restrictions and configuration constraints on the interconnection of application elements.

In a manner analogous to type checking in programming languages, it is desirable to be able to perform limited static checking of compatibility when connecting or transforming elements. Such controls facilitate the construction of correct architectures and help designers focus their attention to more complex issues. SYNOPSIS provides such a mechanism, based on matching element attribute values. The mechanism is described in Section 3.4.

3.2 Language Overview

This section introduces a simple example application that will serve as a guide to the various elements of SYNOPSIS. We will refer to this example again in Chapters 5 and 6, in order to illustrate how the system can integrate a set of heterogeneous software components into a smoothly running executable application.

The example system is a file viewer application that consists of three simpler subsystems:

- a user interface that repeatedly asks users for code numbers of interest
- a filename retrieval subsystem that receives user-supplied code numbers and retrieves matching filenames from a database
- a file viewer subsystem that displays the contents of each retrieved file

Figure 3-1 depicts the architecture of the application, expressed as a SYNOPSIS diagram. Two distinct kinds of entities can be immediately distinguished in the diagram: *Activities*, drawn as rectangles, and *dependencies*, drawn as ovals.

Each of the three application subsystems corresponds to a different activity in the diagram. Activity Select Files is *atomic*, while activities Retrieve Filenames and View Files are *composite*, decomposing to patterns of simpler elements. The entire application, File Viewer, is also defined as a composite activity.

Dependencies encode relationships among the application's activities. *Flow* dependencies connect producers and consumers of data resources. In this application, they indicate that user-supplied code numbers must flow from the user interface to the filename retrieval subsystem. Likewise, retrieved filenames must flow from the filename retrieval to the file viewing subsystem. *Prerequisite* dependencies (labeled Persistent Prereq) indicate that the filename database must be opened (once) before any filename retrieval takes place, and that the file viewer subsystem must be initialized (once) before any file can be displayed.

Activities and dependencies are connected together via ports. *Ports* are an abstract mechanism for representing an element's needs for interaction with the rest of the system. For example, activity Retrieve Filename, has three ports: Two of them are data resource ports, encoding the fact that it needs to receive data resources of type `Integer` (code number), and that it produces data resources of type `String` (filename). The third is a Begin control port, encoding the fact that flow of control into the activity must follow the completion of activity Open DB.

All elements of the language (activities, dependencies, ports) may contain an arbitrary number of additional attributes. *Attributes* are name-value pairs that encode additional information about the element. They are also used to express compatibility restrictions that constraint the connection of elements.

Activity ports are connected to corresponding dependency ports using *wires*, drawn as line segments in the diagram. Every time a new connection is attempted, the system performs a compatibility check, based on unification of corresponding attribute values at the two ends of the connection.

The diagram of Figure 3-1 conveys the structure of the File Viewer application independent of the implementation of its elements. It is a generic (specification-level) view of the application. The goal of the process outlined in Section 2.4 is to be able to generate an executable system by applying a series of transformations to that diagram. From a language perspective, this requires support for implementation-level entities that implement the intended functionality of activities and dependencies.

SYNOPSIS supports such a set of entities, defined as attributes of their respective activities and dependencies. They are called *software components, coordination processes* and *software connectors.*

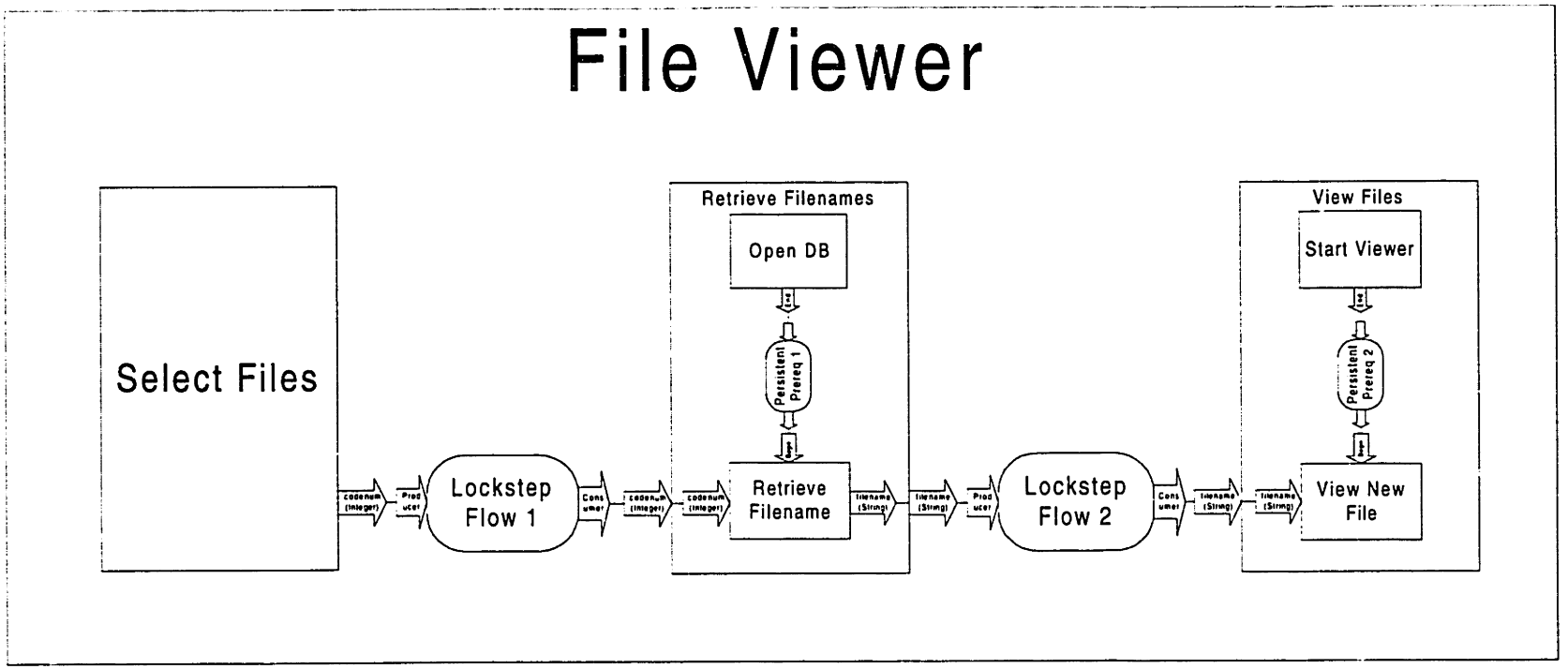


Figure 3-1: SYNOPSIS architectural description of a File Viewer application.

Software components are code-level software entities, such as source code modules, executable programs, user-interface functions, network services etc. SYNOPSIS provides a special notation for describing the properties of software components (see Section 3.3.1.2).

Our objective in this example is to construct the File Viewer application out of a set of existing, heterogeneous code-level components:

- a C source module implementing the user interface subsystem
- a Visual Basic module implementing the filename retrieval subsystem
- a commercial text editor in executable form implementing the file viewing subsystem

Coordination processes manage dependencies. They represent implementations of interaction protocols. They are patterns of simpler activities and dependencies. Finally, *software connectors* represent low-level interconnection mechanisms, directly supported by the semantics of programming languages and operating systems. Examples include procedure calls, method invocations, local variables, etc.

3.3 Language Elements

3.3.1 Activities

Activities represent the main functional pieces of an application. They *own* a set of ports, through which they interconnect with the rest of the system. Interconnections among activities are explicitly represented as separate language elements, called dependencies. Activities must always connect to one another through dependencies. As a consequence, every activity port must be connected to a compatible dependency port.

Activities are defined as sets of attributes which describe their core function and their capabilities to interconnect with the rest of the system. The two most important activity attributes are:

- An (optional) *decomposition*. Decompositions are patterns of simpler activities and dependencies which implement the functionality intended by the composite activity. The ability to define activity decompositions is the primary abstraction mechanism offered by SYNOPSIS.
- An (optional) *component description*. Component descriptions connect SYNOPSIS activities with code-level components which implement their intended functionality.

Examples of code-level components include source code modules, executable programs, network servers, etc.

Depending on the values of the above two attributes, activities are distinguished as follows (Figure 3-2):

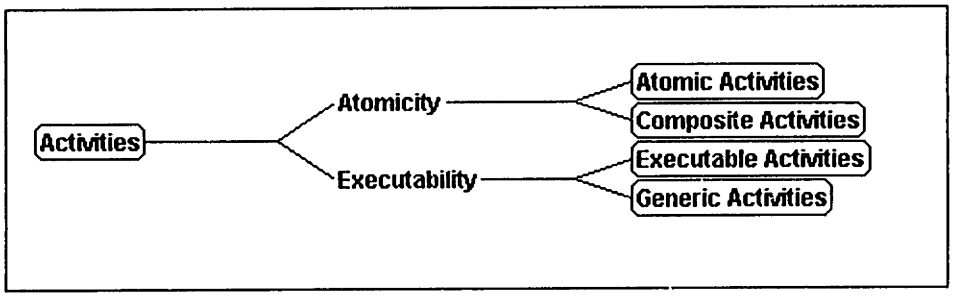


Figure 3-2: A classification of SYNOPSIS activities.

- *Atomic* or *Composite.* Atomic activities have no decomposition. Composite activities are associated with a decomposition into patterns of activities and dependencies.

- *Executable* or *Generic.* SYNOPSIS allows the description of software systems at various levels of abstraction. This is manifested by allowing activities to be *executable* or *generic.* Executable activities are defined at a level precise enough to allow their translation into executable code. Activities are executable either if they are associated with a component description, or if they are composite and every element in their decomposition is executable. Activities which are not executable are called generic. To generate an executable implementation, all generic activities must be replaced by appropriate executable specializations.

Throughout this thesis, we will use the term *primitive* to refer to entities that are both atomic *and* executable.

3.3.1.1 Activity decompositions

Activity decompositions are patterns of activities and dependencies that implement the intended functionality of a composite activity, with which they are associated. Activity decompositions must obey the following structural restrictions (Figure 3-3):

- All *free ports* of activities contained in the decomposition (i.e. ports not connected to other decomposition elements) must be connected to compatible ports of the composite activity. This simply states that the external points of interaction of a composite activity with the rest of the system are equal to the points of interaction of its decomposition, minus the interactions that are internal to the decomposition.

- Dependencies contained in an activity decomposition can have no free ports. This states that, if a dependency is included in a decomposition, all of its ports must be connected to activities included in the same decomposition. Dependencies that span composite activities must be defined outside of any of them. This rule is an equivalent way of expressing the requirement that activities must always be connected to one another through intermediate dependencies.

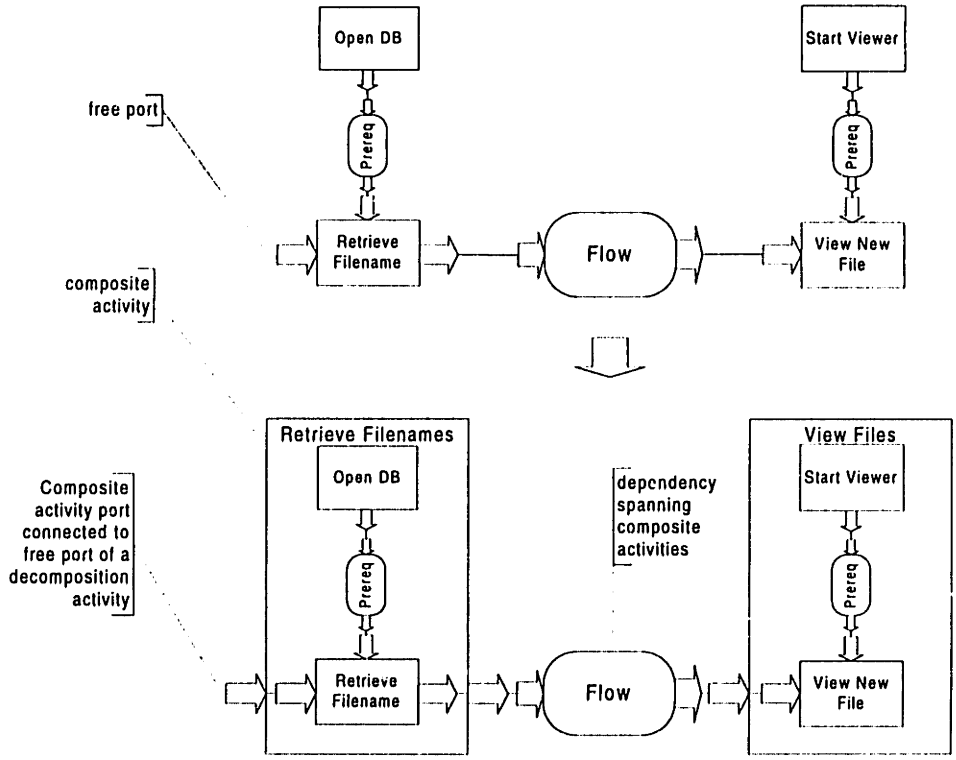


Figure 3-3: Rules for creating activity decompositions.

3.3.1.2 Component descriptions

Activities describe the functional pieces of an application. These functional pieces are implemented either by decomposing them into equivalent patterns of simpler activities and dependencies, or by associating them with appropriate code-level software components. In order to properly integrate such software components into executable systems, our system needs some information about their interfaces, source and object files, and other related attributes. SYNOPSIS provides a special notation called *Component Description Language (CDL)* for describing the properties of software components associated with executable activities. CDL should be thought of as "syntactic sugar" that eventually translates into sets of SYNOPSIS attribute definitions. These definitions are stored in the corresponding executable activity.

SYNOPSIS component descriptions must include information about:

- the component kind
- the provided interface of the component
- expected interfaces of other components
- source and object files needed by the component
- additional attributes specific to the component

The following paragraphs describe each part of the description in more detail:

Component kinds

Software components come in a variety of forms, such as source modules, executable programs, user-interface functions, etc. One of the main objectives of the entire project is to be able to combine together heterogeneous components of arbitrary form. SYNOPSIS provides support for defining arbitrary component kinds. This mechanism is described in detail in Chapter 5. From a CDL perspective, each component kind is represented by a different keyword. Table 3-1 describes the component kinds supported by the current prototype implementation of the system.

Component Type	*CDL Keyword*	*Description*
Source procedure	`proc`	A source code procedure or equivalent sequential code block (subroutine, function, etc.)
Source module	`module`	A source code module, consisting of one or more source code files and containing its own entry point (main program). Modules interact with the rest of the system through expected interfaces only.
Filter	`filter`	A source code procedure that reads its inputs from and/or writes its outputs to sequential byte streams
Executable	`exec`	An executable program
DDE server	`ddes`	A DDE server[1] embedded inside an executable program
OLE server	`oles`	An OLE server[2] embedded inside an executable program
Gui-Function	`gui`	A function provided by the graphical user interface of some executable program, typically activated through a key sequence

Table 3-1: Component kinds supported by the current implementation of SYNTHESIS.

[1] DDE = Dynamic Data Exchange, a Microsoft Windows interapplication communication mechanism.
[2] OLE = Object Linking and Embedding, a Microsoft Windows interapplication communication mechanism [Microsoft94]

Provided component interface

Software components interact with other components by providing input-output interfaces. Depending on the component kind, such interfaces might correspond to procedure parameter lists, executable program command line invocation interfaces, user interface function activation key sequences, etc. When associating software components to executable activities, each element of the provided interface must be mapped to a corresponding atomic port of the activity.

Expected interfaces of other components

This part of the description reflects the fact that current-technology software components often make assumptions about the existence of other components or services in the system. For example, a source module might be calling an externally defined procedure with specified name and para neter interface. A Microsoft Windows executable program might assume the existence of a DDE server application with given specifications. Such expectations are documented in this section, using a syntax identical to the one used to define the component's provided interface. When associating software components to executable activities, each element of each expected interface must also be mapped to a corresponding atomic activity port.

Files needed by the component

Software components are usually pieces of code, residing in one or more source or object files. This section specifies the relevant set of files, to be eventually included in the final application.

Additional component attributes

For every kind of component, there is a set of required attributes that must be specified. Examples of such attributes include the programming language (for source components), the invocation pathname (for executable programs), the service and topic names (for a DDE server component), the activation key sequence (for user-interface services), etc. There are also optional attributes, such as restrictions on the target environment or host machine.

Figure 3-4 shows the CDL descriptions of components associated to the five executable activities of the File Viewer application that was introduced in Section 3.2.

```
Component "Select Files" IsA Procedure

Provides:

    proc select_files();

Expects:

    proc view_selected_files(in codenum:Integer);

Source Files:

    \fviewer\select.c

Attributes:

    Language = c

End Component
```

```
Component "Open DB" IsA Procedure

Provides:

    proc init_DB();

Source Files:

    \fviewer\retrieve.bas

Attributes:

    Language = vb

End Component
```

```
Component "Retrieve Filename" IsA Function

Provides:

    func retrieve_filename(in codenum:Integer):String;

Source Files:

    \fviewer\retrieve.bas

Attributes:

    Language = vb

End Component
```

```
Component "Start Viewer" IsA Executable

Provides:

    exec msword();

Attributes:

    ExePath = \applic\msoffice\winword\winword.exe

End Component
```

```
Component "View New File" IsA Gui-Function

Provides:

    gui open_file(in filename:String);

Attributes:

    guiWindow = "Microsoft Word"
    guiKeys = "^O@1~"

End Component
```

Figure 3-4: Component Descriptions for the File Viewer example application (Figure 3-1).

Activity Select Files is associated with a C source code function, called `select_files`. This function contains an internal loop that repeatedly asks users for a code number. For each user-supplied code number, it calls function `view_selected_files`, to which it passes the user-supplied code number as an argument. Function `view_selected_files` is not part of this module and is *expected to exist* somewhere in the same executable. Its single parameter `codenum` is mapped to a corresponding producer port of activity Select Files. File `select.c` contains the code and data definitions for `select_files` and any other internal functions it may call.

Similarly, activities Open DB and Retrieve Filename are associated with Visual Basic procedures defined in file `retrieve.bas`.

Activity Start Viewer is associated with an executable program (Microsoft Word, a commercial text editor). Enactment of the activity corresponds, in this case, to invoking the program. The component's CDL provided interface describes the command line parameters for starting the program. In this case there are no command line parameters, other than the program pathname.

Finally, activity View New File is associated with a graphical-user-interface function offered by the text editor. In order to display a new file, the text editor normally expects to receive a key sequence in its main window, consisting of the character CTRL-O, followed by the required filename, followed by a newline character. Thus, this service expects one abstract data resource (the filename), embedded in a key sequence whose generic format is described by attribute `guiKeys`. In this example, `guiKeys` contains the characters `^O` (=CTRL-O), followed by `@1` (=the value of the first interface element, in this case the filename), followed by ~ (=newline). Finally, attribute `guiWindow` specifies the title of the editor's main window, where the activation key sequence must be sent.

3.3.2 Dependencies

Dependencies describe interconnection relationships and constraints among activities. Traditional programming languages do not support a distinct abstraction for representing such relationships and implicitly encode support for component interconnections inside their abstractions for components (see Chapter 2). In contrast, SYNOPSIS requires that all interconnections among activities are explicitly represented using dependencies. Like activities, dependencies own ports. Dependency ports must be connected to compatible activity ports.

Like activities, dependencies are defined as sets of attributes. The most important attributes are:

- An (optional) *decomposition* into patterns of simpler dependencies that collectively specify the same relationship with the composite dependency.

- An (optional) *coordination process*. Coordination processes are patterns of simpler dependencies and activities that describe a mechanism for managing the relationship or constraint implied by the dependency.

- An (optional) association with a *software connector*. Connectors are low-level mechanisms for interconnecting software components that are directly supported by programming languages and operating systems. Examples include procedure calls, method invocations, shared memory, etc.

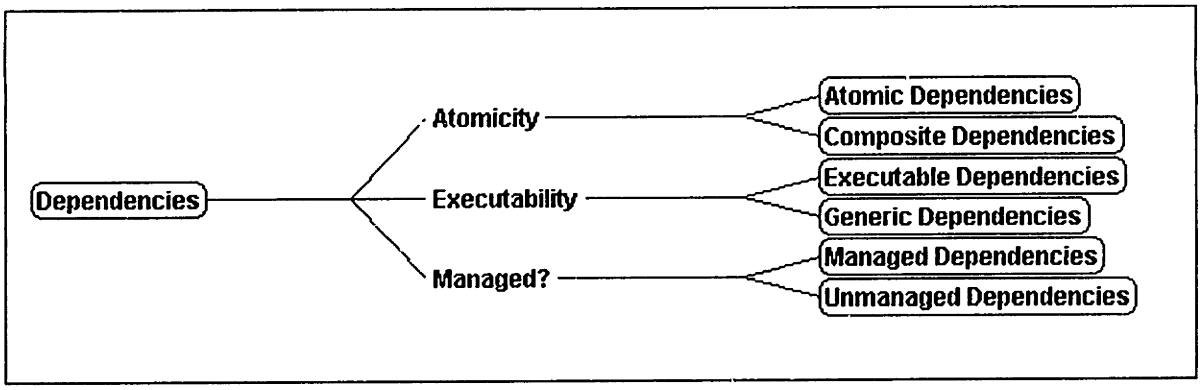


Figure 3-5: A classification of SYNOPSIS dependencies.

Depending on their values in the above attributes, dependencies are distinguished into the following categories (Figure 3-5):

- *Atomic* or *composite*. Atomic dependencies have no decomposition. Composite dependencies are associated with patterns of simpler dependencies that specify the same relationships.

- *Managed* or *unmanaged*. Managed dependencies have an associated coordination mechanism, specifying one way of managing the interconnection relationship they represent. Unmanaged dependencies have no associated coordination process. They simply specify the existence of a relationship and must be specialized by appropriate managed dependencies before code generation can take place.

- *Executable* or *generic*. Executable dependencies are defined at a level specific enough to be translated into executable code. Dependencies are executable if one or more of the following conditions hold: (1) they are directly associated with a code-level connector, (2) they are composite and all elements of their decomposition are executable, (3) they are managed and all elements of their coordination process are executable.

SYNOPSIS allows the definition of arbitrary dependency types. However, one of our goals is to make the step of specifying and managing dependencies a routine one. For that reason, it is useful to define a standardized vocabulary of common dependency types that covers a large percentage of activity relationships encountered in software systems. Such a vocabulary will be described in Chapter 4. Figure 3-6 shows some examples of common dependencies.

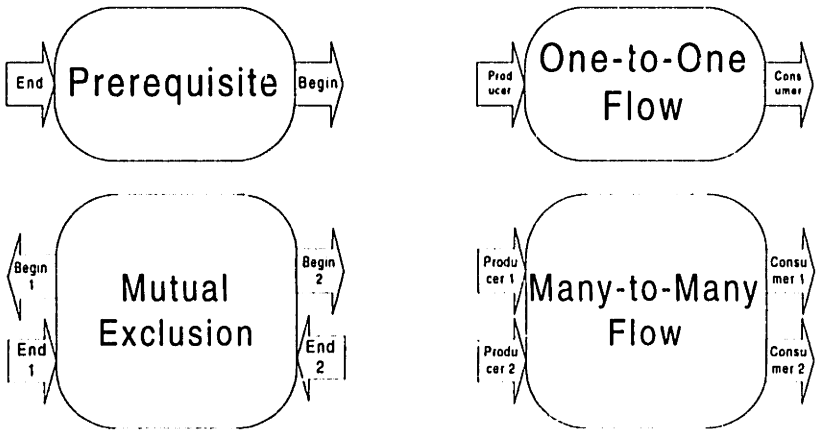


Figure 3-6: Examples of common dependency types.

3.3.2.1 Dependency decompositions

Some dependencies can be equivalently described by patterns of simpler dependencies. Such patterns are called dependency decompositions and are defined as attributes of composite dependencies. Figure 3-7 shows some examples of composite dependencies and their decompositions.

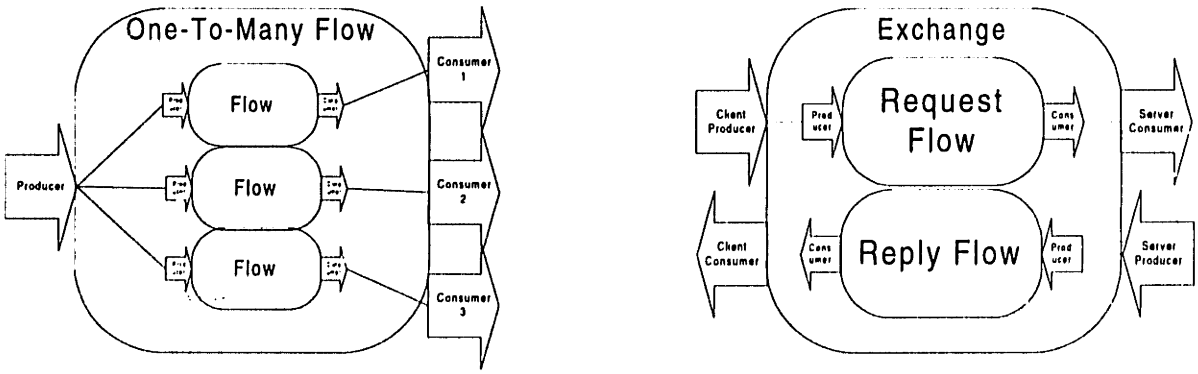


Figure 3-7: Examples of dependency decompositions.

3.3.2.2 Coordination processes

Dependencies specify interconnection relationships which translate into *needs for interaction* among a set of activities. For example, a prerequisite dependency indicates a need to make sure that an activity does not take place before another activity has ended.

To generate executable systems, dependencies must be *managed* by inserting appropriate *interaction* or *coordination* mechanisms into the system. SYNOPSIS provides the abstraction of coordination process, defined as an attribute of dependencies, to represent and localize the definition of such interaction mechanisms.

Coordination processes are patterns of simpler dependencies and activities. In that sense, they are very similar to activity decompositions. Coordination processes must obey the following structural restrictions, which are the dual of the equivalent restrictions for activity decompositions (see Section 3.3.1.1):

- All *free ports* of dependencies contained inside a coordination process (i.e. ports not connected to other elements of the coordination process) must be connected to compatible ports of the associated dependency. This simply states that the external points of interaction of a coordination process with the rest of the system are equal to the points of interaction of its elements, minus the interactions that are internal to the coordination process.

- Activities contained inside a coordination process can have no free ports. This basically states that, if an activity is included in a coordination process, all of its ports must be connected to dependencies included in the same coordination process. Activities that span coordination processes must be defined outside of any of them. This rule is an equivalent way of expressing the requirement that dependencies must always be connected to one another through intermediate activities.

Figure 3-8 shows a SYNOPSIS definition of a coordination process that implements a pipe channel protocol (manages one-to-one flow dependencies). It is a pattern of three activities, and five lower-level dependencies.

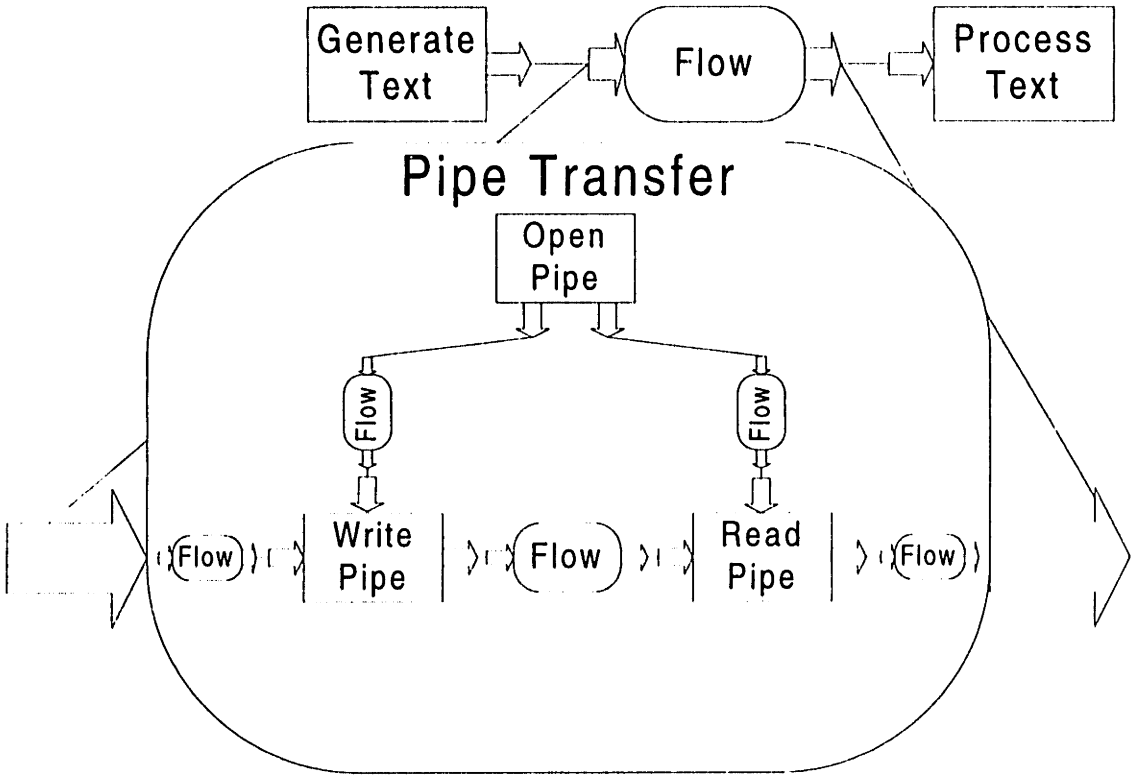


Figure 3-8: A coordination process for managing flow dependencies.

3.3.2.3 Software connectors

Dependencies describe interconnection relationships among components. These relationships are managed either by coordination processes, described as patterns of simpler activities and dependencies, or by associating them with appropriate code-level software connectors. Connectors represent low-level software component interaction mechanisms built into the semantics of programming languages, or implicit in the semantics of operating system calls. Examples include:

- design-time ordering of program statements in the same sequential code block (manages prerequisite dependencies)
- local variable transfer (manages flow dependencies)
- transport of data from one process to another implicit in the semantics of a UNIX pipe protocol

3.3.3 Ports

Ports encode abstract interfaces ("needs for interaction") of activities and coordination processes. All connections between SYNOPSIS language elements are done through ports.

As is the case with activities and dependencies, SYNOPSIS ports are distinguished into *composite* and *atomic.*

- Composite ports act as *abstract port specifications*, describing a port's logical role in the system. They decompose into sets of simpler ports, which might themselves be composite or atomic.

- Atomic ports directly map to an element of some implementation-level component interface (see Section 3.3.1.2).

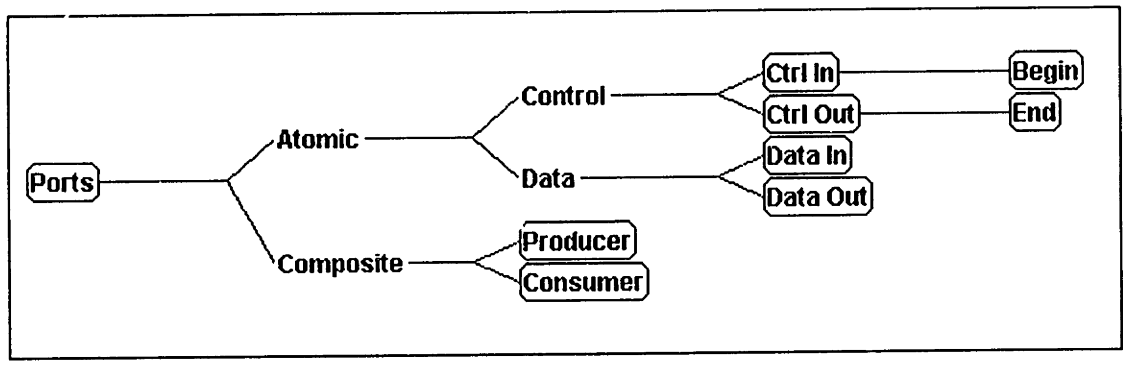


Figure 3-9: SYNOPSIS built-in port specialization hierarchy

Figure 3-9 shows the SYNOPSIS built-in specialization hierarchy for ports. SYNOPSIS users are free to define new ports as specializations of the built-in types, or define completely new port types.

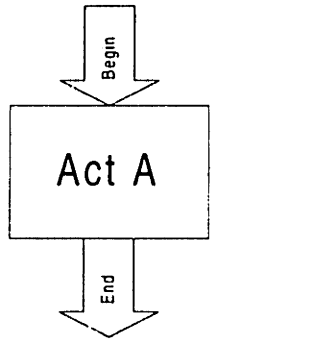
3.3.3.1 Atomic ports

The built-in port hierarchy implies that atomic ports are either input or output ports of data or control.

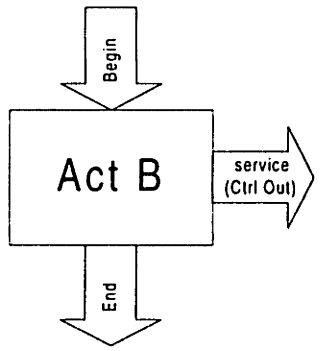
Atomic data ports are mapped to atomic interface elements of software components or executable coordination processes. The mapping is performed by setting attribute `MapsTo` at the port to a string indicating the respective interface element. Depending on the component kind, an atomic port might thus correspond to a source procedure parameter, a command line argument, a user-interface-function activation key sequence,

an event detected or generated by a program, an object generated or used by an executable program, etc. All atomic data ports contain an attribute named `Resource`, which lists the data type expected to flow through the port.

Atomic control ports model the flow of control in and out of an activity or coordination process. Every activity has at least one pair of control ports modeling the beginning and end of execution of the activity. These ports are called the Begin and End ports. Complex activities might have additional control ports, modeling, for example, periodic calls to other activities that take place during the life of the original activity (Figure 3-10).




```
Component "Act A" IsA Procedure
Provides:
    proc act_a();
End Component
```




```
Component "Act B" IsA Procedure
Provides:
    proc act_b();
Expects:
    proc service();
End Component
```

Figure 3-10: Begin, End, and additional control ports.

3.3.3.2 Composite ports

Sets of atomic ports can be optionally grouped inside composite ports. Our built-in port hierarchy contains only two types of composite ports: *producer* and *consumer* ports, specifying the production and consumption of abstract resources.

Designers might want to use composite ports for two reasons:

- to indicate sets of logically related interface elements in a given implementation, and

- to indicate generic needs of interaction when designing generic entities. Different specializations of a given generic entity might contain different decompositions of the same composite port, corresponding to the variety of interfaces by which a given abstract need for interaction can be implemented.

The rest of the section describes an example of the use of composite ports.

We are designing an application which processes the contents of an input file. One of the activities in this application will make the contents of that file available to the rest of the system. Initially, we do not know the exact implementation and interface of the component that will implement that activity. We, therefore, specify a generic activity called Generate Input, which contains a generic producer port, through which the contents of the input file will be made available to the rest of the system (Figure 3-11(a)). In the application diagram, that producer port will be connected to dependency ports connecting it to the consumer activities.

There are at least two ways to implement activity Generate Input. Figure 3-11(b) shows a *filter* implementation. The activity is associated to a component that writes its output to an externally opened sequential byte stream. The same byte stream must be read by consumer activities to retrieve the contents of the input file. In this case, the composite producer port decomposes to a data input port, corresponding to the file descriptor of the sequential byte stream that must be passed to the filter. It is interesting to note that, in this case, a composite producer (i.e. output) port, decomposes into an atomic data *input* port.

Figure 3-11(c) shows another way to implement Generate Input. For each line of the input file, the second implementation calls a procedure, which it expects to have been defined externally. It passes the current input line as a string parameter to the procedure. In this case, the composite input port decomposes into a data output port, corresponding to the string parameter of the procedure called by the component.

By default, every composite activity port includes the Begin and End ports of its associated activity in its decomposition. Activity ports are directly connected to coordination process ports. Coordination processes often need to control what happens before or after the execution of an activity. This requires them to have access to the Begin and End ports of an activity. In actual practice, this requirement is frequent enough that we decided it to include Begin and End ports in the decomposition of every composite activity port by default.

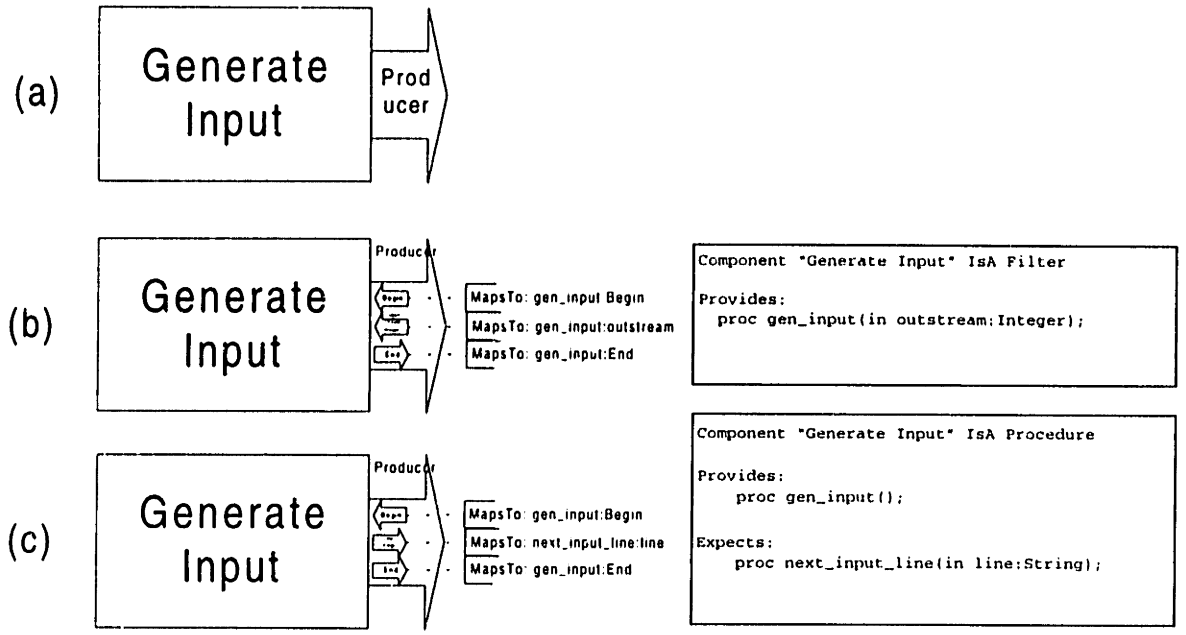


Figure 3-11: *(a) A generic activity containing a generic producer port.*
(b) One executable specialization of the previous activity and its corresponding generic port decomposition into atomic ports.
(c) Another executable specialization of the same activity, which has a different decomposition of the generic port.

3.3.4 Resources

Most interactions among software components center around the flow and sharing of various kinds of resources. In most cases, resources are dynamically produced during run-time, as a result of execution of activities. Such dynamically produced resources are implicitly modeled in SYNOPSIS diagrams through producer ports. As mentioned in the previous section, ports contain special attributes which describe the data type, or other properties, of the resources that flow through them.

In many software applications, however, there exist some resources that have been produced outside of the scope of the application. Examples include hardware resources, such as a printer, and resources produced by other applications, such as a database.

SYNOPSIS models such preexisting resources using a special resource language entity. Like activities, resource entities own ports, through which they are connected to the rest of the system. Atomic activity ports, are associated with interface elements of software components. In contrast, *atomic resource ports* are bound to constant data values, which usually correspond to resource identifiers.

Figure 3-12 depicts an example of the use of resource entities in SYNOPSIS diagrams. Resource entity Database represents a preexisting database file which is being shared by

two activities in this application. Database has a single port, which is bound to the filename of the database file. This filename must be made known to the two user activities (through a Flow dependency), in order for them to be able to access the database.

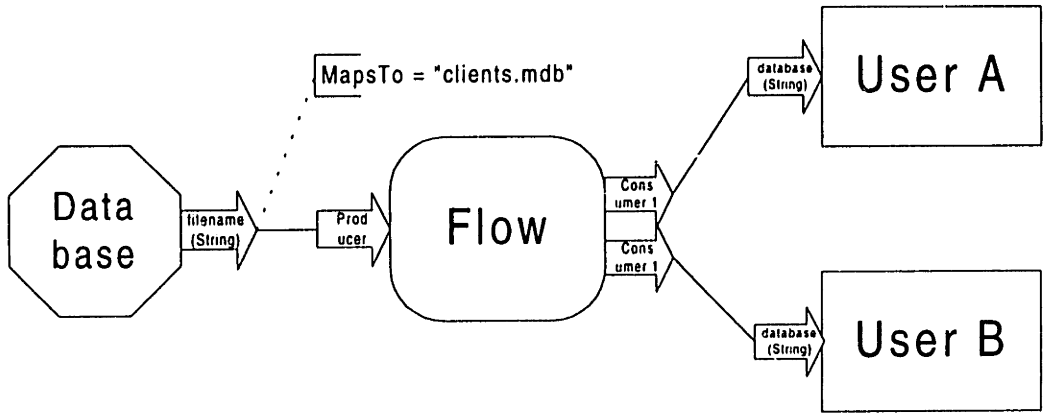


Figure 3-12: Example of resource entity use in SYNOPSIS diagrams.

3.3.5 Attributes

Attributes are name-value pairs that specify additional information about SYNOPSIS language entities. Attribute values can be scalar, lists, or pointers to other SYNOPSIS entities. Although SYNOPSIS supports the definition of arbitrary attributes, the system is using some conventions about required attributes in each entity type. Some of those conventions have been collected in Table 3-2. As more experience is gained from using the system, more such useful conventions are expected to emerge.

In the current implementation attributes play an important role:

- in determining compatibility of an element with its neighbors in the system
- in integrating components and coordination processes into sets of executable modules

The following additional features of the attribute mechanism are especially designed to support the compatibility checking process:

Entity type	*Required Attributes*	*Description*
All activities	`Environment`	execution environment(s) for which activity is compatible

Activities associated to source module components	`Language`
	
	`Executable`
		source programming language of module
		
		name of executable where module is to be integrated

Activities associated with executable program components	`ExePath`	pathname of executable program

Activities associated with DDE server components	`Service`
	
	`Topic`
		Server DDE service name
		
		Server DDE topic name

Activities associated with graphical user interface function components	`guiKeys`
	
	`guiWindow`
		Format string of activation key sequence
		
		Name of window where activation key sequence should be sent

All atomic ports	`MapsTo`
	
	`Resource`
		pointer to interface element corresponding to port
		
		type of data flowing through port

Table 3-2: Partial list of required attributes for each SYNOPSIS entity type.

Attribute Inheritance

As will be described in Section 3.5, all SYNOPSIS language elements inherit the decomposition and all attributes from their specialization parents. In addition to that, SYNOPSIS language elements also inherit all attributes defined at their *decomposition* parents. This feature enables attributes that are shared by all members of a composite element to be defined only once at the decomposition parent level. For example, in Figure 3-13, all activities and dependencies contained inside UNIX Pipe Transfer inherit the attribute definition `Environment = UNIX`, which denotes that they are compatible with the UNIX environment.

Since ports owned by an element are considered to be parts of its decomposition, ports always inherit all attributes defined at their owner.

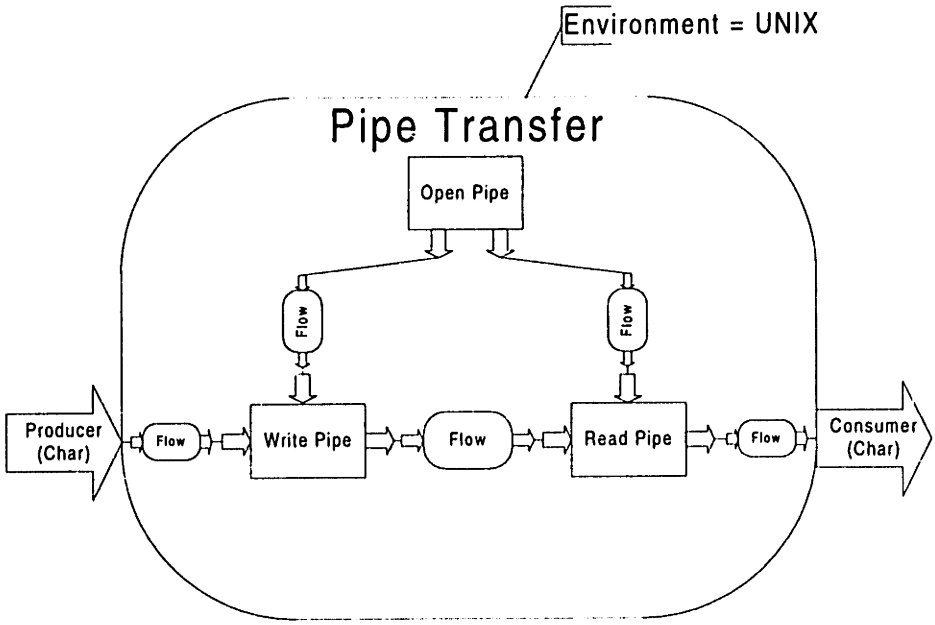


Figure 3-13: Attribute definitions are inherited by all decomposition children.

Attribute Variables

In addition to normal values, attributes can be assigned to attribute variables. Attribute variables are strings that begin with "`?`". For example:

```
Resource = ?res
Procedure = ?proc
```

Attribute variables need not be declared. They are automatically created the first time they are referenced.

Unique variables are special attribute variables whose names begin with the string "`?unq`" (e.g. `?unq-var`, `?unq1`, etc.). They have the special requirement that their values cannot be equal to the value of any other variable in the same scope. They are used to express inequality constraints (see Section 3.4.1).

Attribute variables are local to each owner element (activity, dependency, resource) but are shared among an owner element and all its ports. For example, in Figure 3-14, both references to variable `?res` at ports of Activity A refer to the same variable. Setting the variable at one of the ports will affect both attribute definitions. Likewise, both references to variable `?res` at ports of Activity B refer to the same variable. However, references *across* activities refer to different variables. For example, setting the variable `?res` at Activity A will not affect the value of variable `?res` at Activity B.

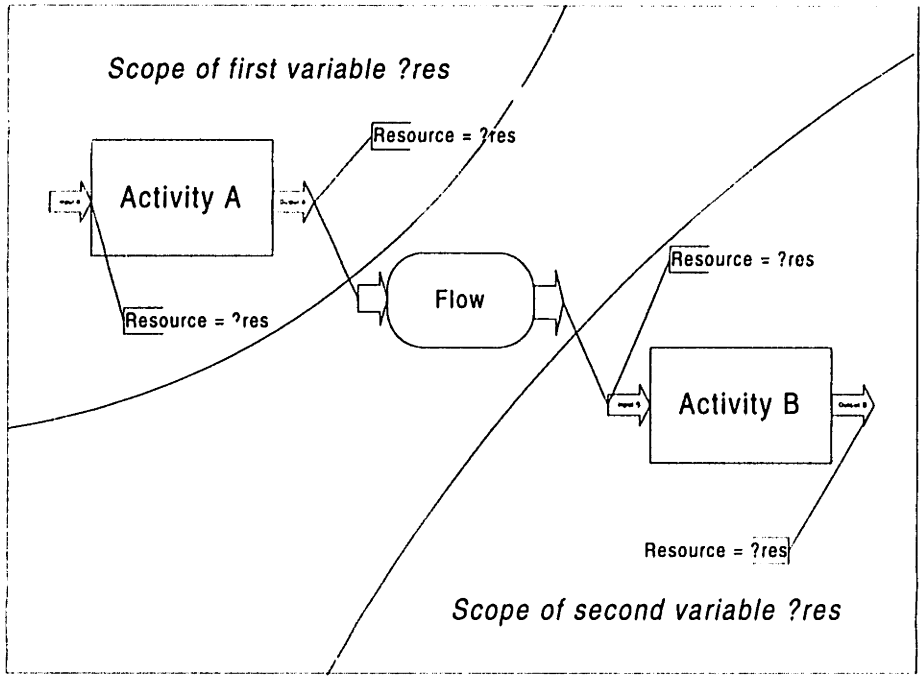


Figure 3-14: Scope rules of attribute variables.

3.3.6 Wires

Wires, drawn as simple line segments, indicate that a legal connection has been established between two ports. Wires established between composite ports imply a set of legal connections between each of their corresponding subports.

There are two types of wires in SYNOPSIS diagrams.

- *Internal wires* connect free ports of decomposition or coordination process elements to compatible ports of their associated composite elements. For example, in Figure 3-3 two internal wires connect the two free ports of atomic activity Retrieve Filename to compatible ports of composite activity Retrieve Filenames. Likewise, in Figure 3-8 internal wires connect the three Flow dependencies contained in the decomposition of One-To-Many Flow to compatible ports of the composite dependency. Establishment of a legal internal wire requires that the decomposition or coordination process port (the *internal* port) is equal to or a specialization of the composite element port (the *external* port) to which it is being connected.

- *External wires* connect activity ports to dependency ports. It is illegal to connect an activity port directly to another activity port, or a dependency port directly to another dependency port. Furthermore, the port at the activity side must be equal to or a specialization of the port at the dependency side. This restriction reflects the coordination perspective on software architecture adopted by SYNOPSIS, whereby

software systems are represented as sets of activities interconnected through explicitly represented relationships (dependencies).

3.4 Checking Element Compatibility

SYNOPSIS language elements have several semantic restrictions and constraints on their ability to connect to other elements. Some of these restrictions are encoded in the syntax of the language. For example, activity ports can only be connected to dependency ports (and vice versa). Furthermore, dependency ports can only connect to activity ports that are equal to or a specialization of them. Such restrictions are described in Section 3.3.6. Other restrictions are specific to the attributes of code-level elements (components and connectors) associated with executable activities and dependencies. Examples of such restrictions include:

- An activity associated with a source code procedure returning an integer value can only be connected to other activities through dependencies associated with coordination processes capable of transporting integer values.

- Flow dependencies managed by local variable connectors can only connect activities whose associated components can be packaged inside the same sequential code block.

- Flow dependencies managed by DDE transfer coordination processes can only connect activities whose associated components will be packaged in different executables within the same Microsoft Windows-based host.

We would like to be able to encode such restrictions and constraints, so that a number of compatibility tests can be performed automatically by the language. Hence, a problem similar to type checking in a programming language arises in SYNOPSIS whenever two elements are connected together. Since all SYNOPSIS connections are between ports, it is desirable to be able to perform compatibility checks at the port level.

SYNOPSIS bases its compatibility checking on attributes. Attributes are used to specify compatibility restrictions. Before establishing a connection between two ports, SYNOPSIS performs a port matching algorithm that determines port compatibility.

Figure 3-15 describes the algorithm. The algorithm compares the values of all attributes that exist at both sides of the attempted connection. If both attributes have a fixed value then the two values must match. If either attribute is assigned to a variable, a process similar to unification takes place. If at least one attribute fails to match, the connection is considered illegal.

```
Check_Compatibility( aport, dport )
--                  aport = activity port
--                  dport = dependency port
-- Returns:         SUCCESS if ports can be legally connected, FAILURE otherwise
-- Uses:            Match_Values( va, vd )

 If one port is composite and the other is atomic then return FAILURE.

 If both ports are composite then recursively match subports.

 If both ports are atomic then
        If aport is same as or a specialization of dport then
                For each attribute defined at both ports (including inherited attributes)
                        If both ends have a value then  call Match_Values
                        else If one end refers to a variable then
                                If variable has a value then call Match_Values
                                else    Set variable and its equivalence class to value at other end
                                        Return SUCCESS.
                        else If both ends refer to variables
                                If one or both variables have values then do as above.
                                If no variable has a value then
                                        Unify both variables into an equivalence class
                                        Return SUCCESS.
        else return FAILURE.

Match_Values( va, vd )
--                  va = value of attribute at activity side
--                  vd = value of attribute at dependency side
-- Returns:         SUCCESS if values match, FAILURE otherwise.

 If values are identical then return SUCCESS.
 If values are pointers to language elements then
        If va is a specialization of vd[1]
                then return SUCCESS.
 Return FAILURE
```

Figure 3-15: Port compatibility checking algorithm.

3.4.1 Using Attributes to Encode Constraints

The semantics and scope rules of attributes and variables (see Section 3.3.5) permit a number of compatibility constraints to be encoded into attributes. The following paragraphs give some examples of how some of the most useful kinds of constraints can be encoded using the attributes mechanism.

[1] Exception: When comparing resources of consumer ports the opposite specialization relationship must hold.

"An attribute must have a specific value at all neighbor elements"

This is encoded by defining the attribute with the specified value at all ports where this condition is required. If the condition is required at all ports of an owner element, the attribute need only be defined once at the owner element.

Example: A dependency managed by a coordination process encoding a UNIX pipe protocol can only be used to connect components that will run in UNIX environments. Designers can encode this constraint by defining an attribute `Environment` set to `UNIX` at the top level of the process (Figure 3-13). This attribute is inherited by all ports. In order to pass the compatibility check, all connected activities must contain the attribute set to the same value.

"An attribute must have the same (unspecified) value at all neighbor elements"

This can be encoded by setting the attribute to the same variable name at all ports where this condition is required. The unification process will only succeed if the equivalent attribute at the other end of all connections has the same value.

Example: Local variable transfer is the simplest coordination process for managing flow dependencies. However, it can only be used if all its members (a) produce and use the same data type, and (b) can be packaged into the same procedure. The latter requirement implies that they also have to be written in the same language and be packaged into the same executable program. All these constraints can be encoded into the ports of the associated dependency as shown in Figure 3-16.

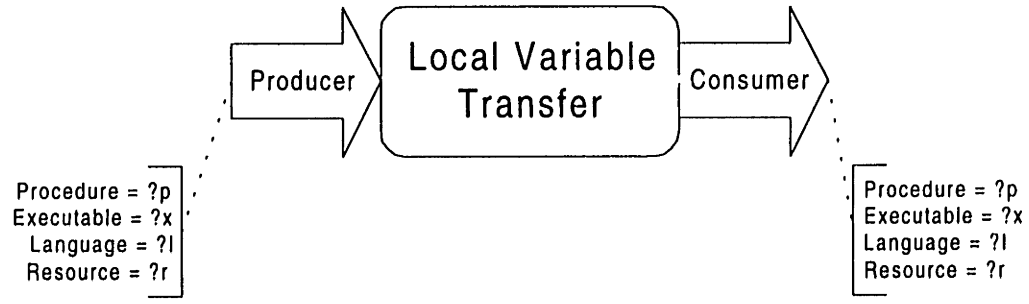


Figure 3-16: Activities connected to both endpoints of this dependency must have identical values for the specified attributes.

"An attribute must have a different value at each neighbor element"

This can be encoded by setting the attribute to a different unique variable (see Section 3.3.5) at each port where this condition is required. Since each unique variable cannot be unified to a value given to any other variable, unification will only succeed if the attribute has a different value at each neighbor element.

Example: Dynamic Data Exchange (DDE) is one protocol for transferring data among different executable programs in the Microsoft Windows operating system. It does not work properly for transferring data within the same executable. Thus, it can only connect components that will be integrated into different executables. This constraint can be encoded into the ports of the associated dependency as shown in Figure 3-17.

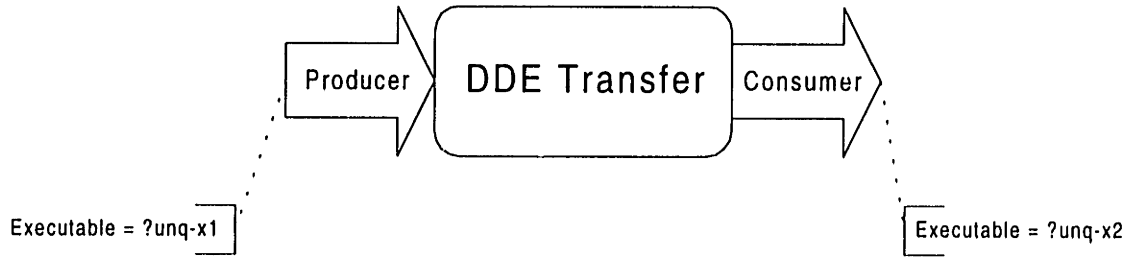


Figure 3-17 : Activities connected at the endpoints of a dependency managed by a DDE Transfer coordination process must be packaged into different executables.

3.4.2 The Dual Purpose of Attribute Matching

Attribute unification simultaneously checks constraint compliance and creates equivalence classes of attribute values across a number of elements. For example, an attempt to connect two activities through a dependency managed by a local variable transfer coordination process (Figure 3-16), not only checks whether the two endpoint activities can be packaged together into the same procedure, but also unifies attribute `Procedure` at both of them to the same equivalence class. This will help the code generator determine that the two activities should be packaged into the same procedure.

Attribute unification is therefore one of the mechanisms used by the system in order to integrate activities and dependencies into sets of executable modules. This aspect of the system will be described in more detail in Chapter 5.

3.5 Entity Specialization

Object-oriented languages provide the mechanism of inheritance to facilitate the incremental generation of new objects as specializations of existing ones, and also to help organize and relate similar object classes. SYNOPSIS provides an analogous mechanism called *entity specialization*. Specialization applies to all the elements of the language, and allows new activities, dependencies, ports and resources to be created as special cases of existing ones. Specialized entities *inherit* the decomposition and other attributes of their parents. They can differentiate themselves from their *specialization parents* by modifying

their structure and attributes using the operations described below. Entity specialization is based on the mechanism of process specialization that was first introduced by the Process Handbook project [Malone93, Dellarocas94].

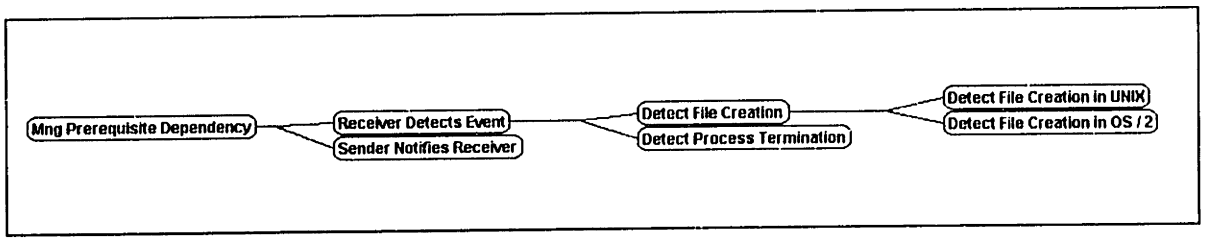


Figure 3-18: A hierarchy of increasingly specialized coordination processes for managing prerequisite dependencies.

The mechanism of entity specialization enables the creation of specialization hierarchies for activities, dependencies, ports, and coordination processes. Such hierarchies are analogous to the class hierarchies of object-oriented systems. In specialization hierarchies, generic designs form the roots of specialization trees, consisting of increasingly specialized, but related designs. The leafs of specialization trees usually represent design elements that are specific enough to be translated into executable code (Figure 3-18). Specialization hierarchies have proven to be an excellent way to structure repositories of increasingly specialized design elements, such as the vocabulary of common dependency types we shall introduce in Chapter 4.

Apart from enabling the organization of related design elements in concise hierarchies, entity specialization also encourages application development by incremental refinement of generic application architectures. Designers can initially specify their applications using generic activities and dependencies. Such generic application architectures can then be iteratively refined by replacing their generic elements with increasingly specialized versions, until all elements of the architecture are executable. The existence of repositories of increasingly specialized design elements can reduce the specialization steps to a routine selection of appropriate repository elements.

3.5.1 Creating New Elements

In a way analogous to object-oriented class creation, all SYNOPSIS elements are created as specializations of other elements. New elements inherit the decomposition, coordination process, and all other attributes defined in their specialization parent. They can differentiate themselves from their parents by applying any number of the following transformations to their attributes (see [Wyner95b]):

- Add or delete a decomposition or coordination process element

- Replace a decomposition or coordination process element with one of its specializations

- Add or delete a connection

- Add or delete an attribute

- Modify the value of an attribute

Figure 3-19 shows an example of how an executable dependency can be defined as a special case of a generic one by inheriting the coordination process of its parent and then replacing all generic activities and dependencies contained therein with executable specializations. Figure 3-19 (a) depicts a generic coordination process for managing flow dependencies using pipes. The process describes the essence of a pipe protocol independently of the particular system calls that are used to implement it in a given environment. It is generic because it contains three generic activities and five unmanaged dependencies. Figure 3-19 (b) depicts an executable specialization of the pipe transfer protocol, specific for the UNIX operating system. The specialization inherited the design elements of the original protocol and replaced generic activities and unmanaged dependencies with executable specializations, appropriate for the UNIX operating system.

3.5.2 Specializing Activities

This operation allows incremental refinement of architectural diagrams. Designers can specify SYNOPSIS architectures in very generic terms and then incrementally specialize their elements until an executable design has been reached. Such a process is supported by the ability to maintain hierarchies of increasingly specialized activities, encoding families of alternative designs for specific parts of software architectures.

From a linguistic perspective, an activity can only replaced with a specialization if the new specialization is able to structurally replace the original activity and legally connect to all neighbor elements connected to it. If the specialization is composite, the same requirements apply for its decomposition as well. The requirements translate to the following port compatibility requirements:

- For each connected port of the original activity, there must be an equivalent port of the specialization.

- Each port of the specialization must be compatible with the dependency port to which the corresponding port of the original activity was connected.

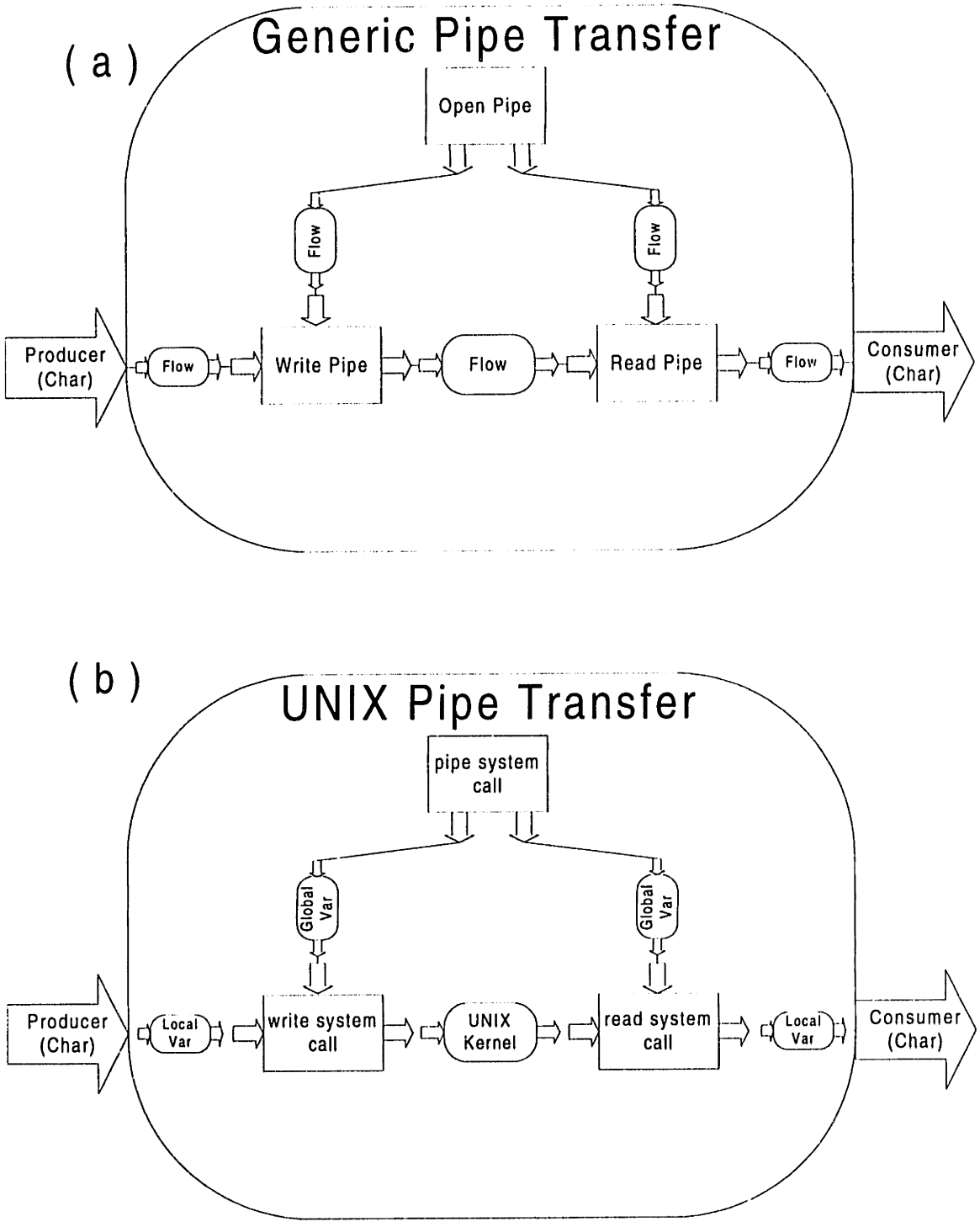


Figure 3-19: *(a) A generic coordination process that manages flow dependencies using pipes. (b) An executable specialization of the previous process.*

3.5.3 Specializing Dependencies

Generic dependencies are descriptions of needs of interaction among software activities. They are useful for constructing configuration-independent architectural descriptions of software systems that clearly show *why* activities need to interact with one another. In order to derive executable implementations from such abstract descriptions, designers can follow two alternative paths:

- If the generic dependency has a decomposition, replace all elements of the decomposition with executable specializations.

- Replace the generic dependency with a managed specialization. Managed dependencies are associated with coordination processes. Coordination processes represent interaction protocols. From a linguistic perspective, a dependency can only replaced with a specialization if the new specialization is able to structurally replace the original dependency and legally connect to all neighbor elements connected to it. If the specialization is managed, the same requirements apply for its coordination process as well. The requirements translate to port compatibility requirements similar to the ones for specializing activities:

  – For each connected port of the original dependency, there must be an equivalent port of the specialization.

  – Each port of the specialization must be compatible with the activity port to which the corresponding port of the original dependency was connected.

3.6 The Way Ahead

SYNOPSIS provides the abstractions needed in order to describe application designs as sets of activities connected by dependencies. It also provides the mechanism of entity specialization that enables generic application architectures to be transformed to executable designs by:

- replacing generic activities with executable specializations

- replacing generic dependencies with managed specializations

- integrating activities and dependencies into executable modules

The first transformation requires locating appropriate code-level components that implement activity functional requirements. We consider this problem the responsibility of the designer and have argued in Section 1.1 that it is being made simpler by today's technologies.

Chapter 4 will present a vocabulary of common dependency types and a design space of associated coordination processes that aims to reduce the second transformation to a routine design step, capable of being assisted, or even automated, by computer.

Finally, Chapter 5 will discuss the problems and solutions of automating the third transformation.

Chapter 4

Towards a Design Handbook for Integrating Software Components

Chapter 2 argued for separating the core functional pieces of a software application from their interconnection relationships. Last chapter introduced an architectural language that enables this separation by providing separate abstractions for activities and dependencies. This chapter goes one step further: It observes that, when taken out of context, many interconnection problems in software applications are related to a relatively narrow set of concepts, such as resource flows, resource sharing, and timing dependencies. These concepts are *orthogonal* to the problem domain of most applications, and can therefore be captured in an application-independent vocabulary of dependency types. Likewise, the design of associated coordination processes involves a relatively narrow set of *coordination concepts*, such as shared events, invocation mechanisms, and communication protocols. Therefore, it can also be captured in a design space that assists designers in designing a coordination process that manages a given dependency type, simply by selecting the value of a relatively small number of design dimensions. The proposed vocabulary of dependencies and design space of coordination processes, taken together, can form the basis for a *design handbook for integrating software components.* The development of such a handbook aims to reduce the specification and implementation of software component interdependencies to a routine design problem, capable of being assisted, or even automated, by computer tools.

4.1 Motivation

The purpose of this chapter is to give an answer to the following two questions:

- Why do software components interconnect with one another?

- How do software components interconnect with one another?

We would like to organize the answers to the first question in a vocabulary of *dependency types*, and the answers to the second question in a design space of *coordination processes*. Finally, we would like to connect each of the *whys*, with a set of *hows*, that is, associate each dependency type with a set of coordination processes for managing it.

A vocabulary of interdependency patterns would greatly aid designers in constructing application architectural diagrams. Instead of always inventing a new dependency to express a given component relationship, designers would often simply choose one from the dependency vocabulary.

Furthermore, the existence of a coordination process design space would reduce the step of managing dependencies with coordination processes to a routine, or even automatic, selection of an element from a coordination process repository.

Finally, a vocabulary of dependency types and coordination processes would contribute to an increased understanding of the problems of software interconnection. Over time, researchers have developed a vast arsenal of algorithms and techniques for process synchronization, communication, and resource allocation. What has been missing so far is a unified framework for relating those algorithms to the problems they are attempting to solve. Such a framework should encompass (and relate to one another) synchronization, communication, and resource allocation considerations. It should relate techniques and algorithms that are currently being studied by a number of different research areas (programming languages, operating systems, concurrent and distributed systems). Therefore, it could form the basis for developing *a design handbook of software component interconnection*. Such a handbook could help reduce the integration of existing software components into new applications to a routine design problem.

The approach taken in this chapter is based on coordination theory [Malone94], an emerging research area that focuses on the interdisciplinary study of coordination. Coordination theory defines coordination as the process of managing dependencies among activities. One of its objectives is to characterize different kinds of dependencies and identify the coordination processes that can be used to manage them. This work extends the frameworks presented in [Malone94], and is the first detailed application of the theory to the understanding of software component relationships. Coordination theory is discussed in more detail in Section 7.2.

It is important to emphasize that the results described in this chapter do not claim rigorous generality and completeness. Our goal was to develop a dependency vocabulary and coordination process design space that covers a *useful* subset of the component relationships and constraints encountered in practice. The SYNOPSIS machinery enables designers to incrementally enrich this vocabulary with new abstractions and processes. It

is our hope that this work will provide a useful starting point that will lead in interesting extensions by future research.

4.2 Overview of the Dependencies Space

The vocabulary of dependencies presented in this chapter is based on the simple assumption that component interdependencies are explicitly or implicitly related to patterns of resource production and usage. *In other words, activities need to interact with other activities, either because they use resources produced by other activities, or because they share resources with other activities.*

The definition of resources can be made broad enough to make this assumption cover most (if not all) cases of component interaction encountered in software systems. Our current definition of resources encompasses:

- processor time (control)
- data of various types
- operating system resources (memory pools, pipes, sockets, etc.)
- hardware resources (printers, disks, multimedia adapters, etc.)

In every resource relationship, participating activities can be distinguished into two roles:

- Resource producers
- Resource consumers

The existence of two different roles in resource relationships, implies the existence of three different classes of dependencies:

- Dependencies between producers and consumers
- Dependencies among consumers who use the same resources
- Dependencies among producers who produce for the same consumers

Dependencies between producers and consumers are modeled using a family of dependencies called *flow dependencies*. Malone and Crowston [Malone94] have observed that, in general, whenever flows occur, one or more other sub-dependencies are present. In particular, flow dependencies can be decomposed to the following set of lower-level dependencies:

- *Usability.* Users of a resource must be able to effectively use the resource. For data resources, this dependency encompasses issues relating to format conversion, semantic equivalence, etc.

- *Accessibility.* In order for a resource to be used by an activity, it must be accessible by that activity (more precisely, it must be accessible by the processor that executes the activity). This requirement might require physical transportation of a resource, or, conversely, relocation of an activity.

- *Prerequisite.* A resource can only be used after the producer activity has been completed. Producers must notify users, or, conversely, users must be able to detect when production is complete.

- *Resource Sharing.* When more than one activity requires usage of a resource, some protocol must manage how the resource will be shared among them. For example, if concurrent access of a resource is not permitted, some kind of mutual exclusion protocol must be put in place.

- *User Sharing.* When more than one producers are producing for the same users, a dependency analogous to resource sharing exists among them. Users become a shared "resource" for producers and some protocol must manage how they are "shared" among multiple producers. For example, users might not be able to use more than one, out of many, resources directed to them. In such cases selection among producers might have to take place.

Sections 4.5 and 4.6 are devoted to a detailed discussion of flow dependencies.

Sharing dependencies contained inside flows assume that multiple users of a resource are independent, and therefore *competing* with one another for resource access. In many applications however, users (or producers) of a resource are *cooperating* in application-specific ways. In those cases, designers must *explicitly* specify additional dependencies that describe the patterns of cooperation among users (or producers). Imagine, for example, a database resource which is generated by some activity and subsequently used by three other activities. In one particular application, one of the users of the database is using it to write values that will be read by the other users. This application-specific pattern of cooperation among users of the database requires the specification of an additional prerequisite relationship between the writer and the reader activities (Figure 4-1).

Application-specific patterns of cooperation among activities that share resources are expressed using additional flows and another family of dependencies called timing dependencies. *Timing dependencies* express constraints on the relative flow of control among a set of activities. The most widely used are *prerequisite dependencies* (A must complete before control flows into B) and *mutual exclusion dependencies* (A and B cannot execute at the same time).

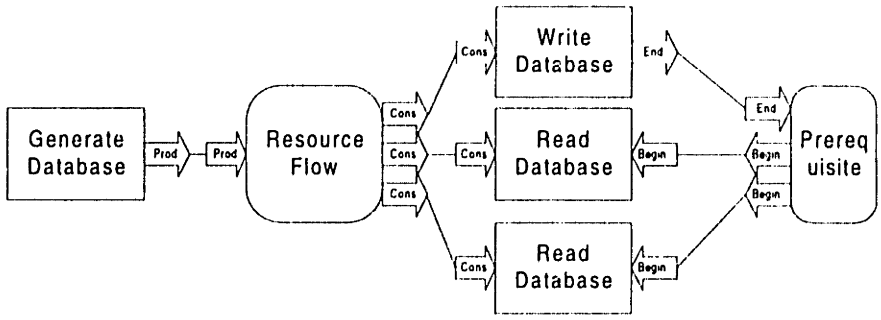


Figure 4-1: An example of cooperative resource use.

In addition to specifying application-specific cooperation patterns, timing dependencies are often used to specify *implicit resource relationships.* For example, mutual exclusion dependencies are often used to specify implicit resource sharing relationships, in which support for resource accesses is embedded inside the code of each activity. Also, prerequisite dependencies often specify implicit flow relationships in which resource production and consumption are embedded inside the code. Section 4.7 describes a family of timing dependencies. For each dependency, its relationship with a resource dependency is illustrated.

Throughout the chapter, it becomes apparent that, apart from classifying and enumerating elementary dependency types, it is also useful to begin to collect and classify sets of frequently occurring *composite dependency patterns.* In many cases, designers have developed specialized, more efficient *joint* coordination processes for such patterns. Section 4.8 will present a few useful composite patterns of flows and joint coordination processes for managing them.

4.3 The Concept of a Design Space

As with any complex taxonomy, it is useful to classify both dependencies and coordination processes using multi-dimensional *design spaces* [Bell72, Lane90]. Each dimension of the design space describes variation in some design choice. Values along a dimension are called *design alternatives.* They correspond to alternative requirements or implementation choices. For example, when selecting a data transportation mechanism, the number of data readers could be one design dimension; the location of readers relative to the writer could be another. Figure 4-2 illustrates a tiny design space for selecting a data transportation mechanism.

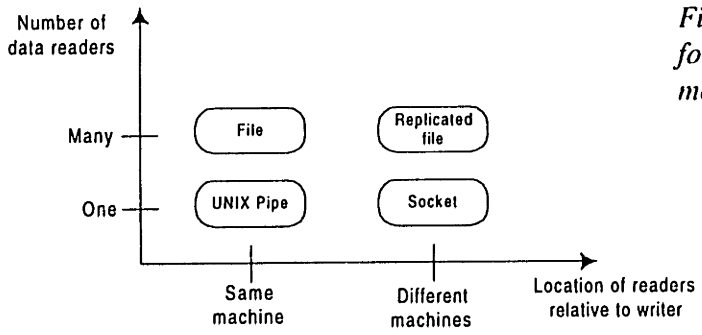


Figure 4-2: A simple design space for selecting a data transportation mechanism.

Specific designs are described by points in the design space, identified by the dimensional values that correspond to their design choices.

Successful design spaces reduce the problem of design to that of answering a simple set of questions. They also organize related design alternatives "close" to each other and expose correlations between various aspects of design. Finally, they can be easily translated into computerized knowledge bases that can help semi-automate the design task.

Our problem requires the construction of two, related, design spaces:

- *A dependency design space.* Dependency design dimensions represent interaction requirements that are significant for choosing a coordination processes. For example, the number of users of a resource and the degree of concurrency allowed by a resource are two dependency design dimensions.

Each point in the dependency design space defines a different *dependency type.*

- *A coordination design space.* Coordination design dimensions represent design alternatives available to satisfy interaction requirements. For example, the protocol used to share a nonconcurrent resource is an implementation design dimension.

Each point in the coordination design space defines a different *coordination process.*

In addition, each point in the dependency design space (dependency type) must be associated to a coordination design space for managing it.

Our objective in the following sections is to define related dependency and coordination design spaces for each family of dependencies.

The success of a design-space description of design alternatives clearly lies in the choice of dimensions and specific dimensional values (design alternatives). There is no obvious rigorous way of defending a particular set of choices. Neither Bell and Newell [Bell72],

nor Lane [Lane90] have offered any justification for their dimensions and alternatives, except for their own intuition and the usefulness of the resulting description. I will follow the same path, simply proposing a set of dimensions and trying to show empirically that they form a useful description of design alternatives.

4.4 A Taxonomy of Resources

Before we begin the description of resource flow dependencies, we present a taxonomy of resources occurring in software systems. This taxonomy will be useful both for distinguishing between different special cases of flow relationships, and for determining the range of alternative ways of managing them.

The taxonomy is summarized in Figure 4-3 The following is a discussion of its principal dimensions.

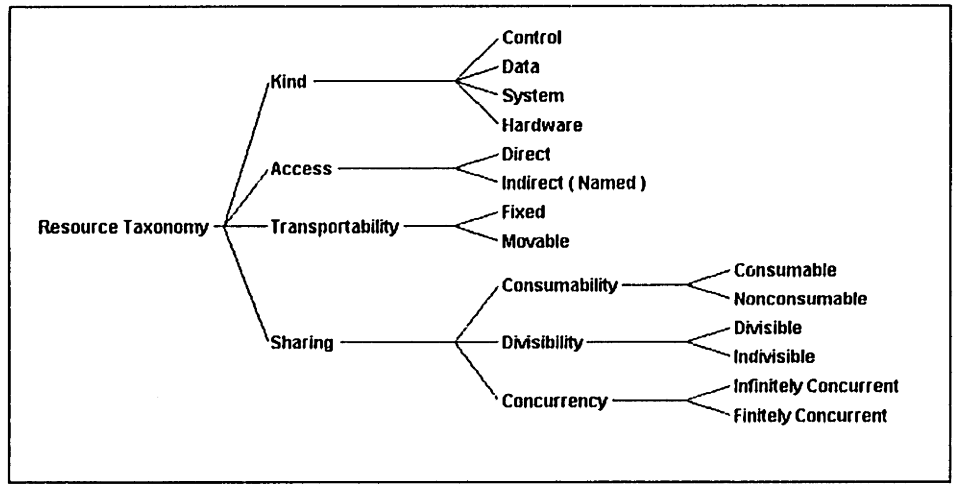


Figure 4-3: A taxonomy of resources.

4.4.1 Resource Kind

- *Control.* The resource usually referred to in Computer Science as *control,* is more accurately described as *a thread of processor attention.* In order for any software activity to begin execution, it needs to receive control *from somewhere*, that is, it needs to receive the attention of some processor. Control flow dependencies thus describe the flow of processor attention from one activity to another.

- *Data.* Data resources include data values such as integers, strings, arrays, etc. They are further distinguished by their data type.

- *System.* System resources represent various services offered by operating systems. They include *passive* resources such as shared memory pools, pipes, communication sockets, and *active* resources, such as name servers, remote file transfer servers, etc.

- *Hardware.* Hardware resources correspond to hardware devices, such as printers, disk and tape drives, multimedia adapters, etc.

4.4.2 Resource Access

Resource access determines how producers and users access their corresponding resources.

- *Direct access resources.* Control and simple data resources are communicated directly from producer to users. In a sense, they are their own identifiers.

- *Indirect (Named) access resources.* Indirect access resources are accessed using a secondary data resource called the *resource name* or *identifier*. Flows of indirect access resources involve the communication of identifiers, rather than the resources themselves.

The use of identifiers is extremely widespread in software systems. Identifiers provide mappings that allow a wide variety of resources (system, hardware, complex data structures) to be accessed by software components that can only interface with their environment through relatively simple data resource ports.

System and hardware resources are always accessed indirectly. Complex data resources, such as files and databases, are also typically accessed using identifiers.

4.4.3 Resource Transportability

Transportability determines whether resources can be moved around in the system.

- *Fixed* resources cannot be moved. They have a fixed location in the system and, in order to use them, software activities have to be located "close" to them. Hardware resources, such as printers, are examples of fixed resources.

- *Movable* resources can be made accessible to other activities by transporting them to other locations in the system. Transportation of a movable resource usually involves an additional auxiliary resource called the *carrier resource.* Data resources are usually movable. For example, a data structure can be moved from one process to another by converting it into a byte stream and transmitting it through a pipe. The pipe (classified as a system resource) acts as the carrier resource in this case.

4.4.4 Resource Sharing

This section describes a framework for reasoning about shared resources that was developed by George Wyner and Gilad Zlotkin at the MIT Center for Coordination Science [Wyner95a].

Wyner and Zlotkin proposed a small number of important resource attributes that can help designers classify coordination requirements for shared resource dependencies. They observed that these important attributes are not merely a function of the resource type, but of the intended *mode of usage* as well. That is, the same resource type, used in different modes (e.g. read versus written), might display different sharing behavior along those attributes. For that reason they refer to them as attributes of *resources-in-use*. These attributes are:

- Divisibility
- Consumability
- Concurrency

a. Divisibility

Divisibility specifies whether a resource-in-use can be divided into independent subresources. Some example of divisible and indivisible resources are shown below.

Resource	*Usage*	*Description*
Divisible		
Memory heap	Read/Write	Heaps can be divided into independent smaller blocks
Network channel	Connect	Physical network channels can support multiple independent connections
Indivisible		
Scalar variable	Read/Write	Scalar variables can only store one value
pgp Encrypted File	Decrypt	Encrypted files can only be decrypted in their entirety

b. Consumability

Consumability specifies whether a resource-in-use is being destructively consumed. Consumable resources can be used a finite amount of times. Nonconsumable resources can be used an arbitrarily large amount of times. Some example of consumable and nonconsumable resources in use are shown below:

Resource	*Usage*	*Description*
Consumable		
Pipe channel	Read	Values "disappear" from the channel as they are being read
PROM	Write	PROMs (Programmable Read Only Memories) can only be written once
Nonconsumable		
File	Read	Files can be read an arbitrarily large amount of times
Processor	Start Task	Processors can be used to start an arbitrarily large number of tasks

c. Concurrency

Concurrency specifies whether a resource-in-use can be used by more than one users at the same time. Concurrency can be finite, setting a finite limit on the number of concurrent users, or infinite (arbitrarily large). The following are examples of finitely and infinitely concurrent resources.

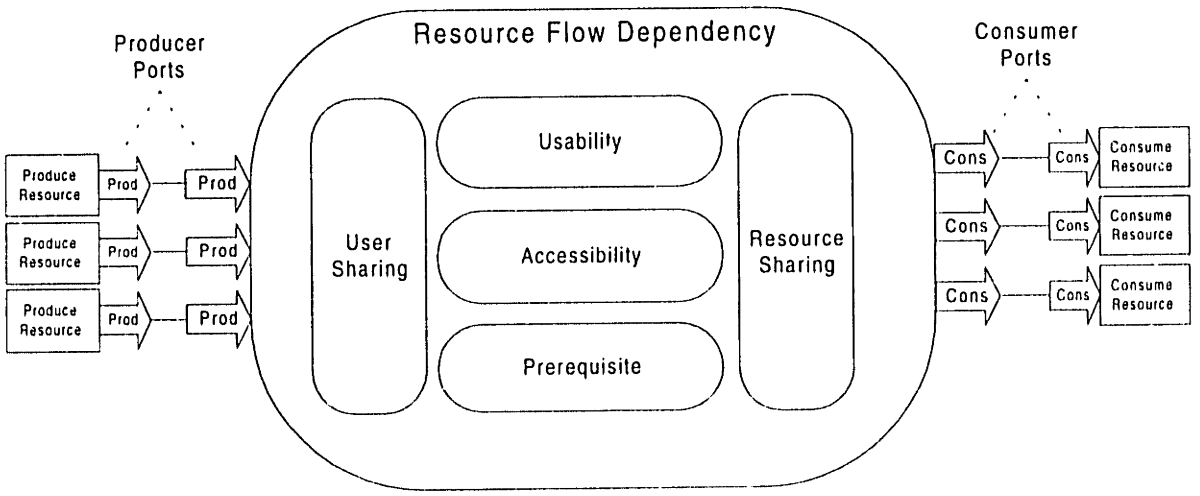
Resource	*Usage*	*Description*
Infinitely Concurrent		
File	Read	In most systems, multiple users are allowed to read files concurrently
Multitasking Processor	Start Task	Multitasking systems appear to execute multiple tasks concurrently
Finitely Concurrent		
Ftp Server	Connect	Ftp servers often limit the number of concurrent connections for performance reasons
Printer	Print File	Printers cannot interleave the printing of different files

4.5 A Generic Model of Resource Flows

This section presents a generic model for classifying flow dependencies and a generic process for managing them. Section 4.6 describes how this generic model can be specialized to manage different special cases of flow dependencies.

In the most general case, flow dependencies exist between a number of resource producers and a number of consumers (Figure 4-4). Dependencies are connected to activities through resource producer and consumer ports. Producer and consumer ports are *abstract ports* (see Section 3.3.3). That is, they are composite ports that contain implementation-specific groupings of low-level interface ports that logically participate in the production and consumption of a given resource.

We assume that, by default, a flow dependency between a set of activities implies a *stream* of resource flows over the lifetime of an application execution. Coordination processes for managing flow dependencies are designed with this assumption in mind. Situations where resources are produced or consumed only once during the lifetime of an application execution are represented by special types of dependencies.



Dependency	*Description*
Usability	Produced resources must be in a form usable by each user
Accessibility	Produced resources must be made accessible to each user
Prerequisite	Resources must be produced before they can be used
Resource sharing	Multiple consumers share the same resources
Consumer sharing	Multiple producers produce resources for the same consumers

Figure 4-4: A generic model of resource flow dependencies.

Our objectives in this section are the following:

- Introduce a set of dependency design dimensions that define a flow dependency design space. These dimensions represent the interaction requirements that are significant in choosing a coordination process. Every point in this design space defines a different special case of a flow dependency.

- Introduce a set of coordination design dimensions that define a coordination process design space. Every point in this design space will be an alternative implementation of a flow dependency managing process.

- Specify how each dependency type restricts the range of possible coordination processes for managing it.

Both design spaces are based on a generic model for decomposing flow dependencies into lower-level dependencies, shown in Figure 4-4. Managing a flow dependency implies managing all lower-level dependencies. This model extends the ideas introduced in [Malone94], and attempts to capture the different considerations that must be addressed whenever resources are exchanged or shared among different activities.

The generic model for managing flow dependencies focuses on the relationships between producers and users of resources. It assumes that different consumers (producers) are independent from one another and compete for access to resources (consumers).

In the following sections, we will first introduce dependency and coordination processes design spaces for each of the lower-lever dependencies. The design space for generalized flow dependencies will then be defined by the product of the component dependencies design spaces.

4.5.1 Usability Dependencies

4.5.1.1 Types of usability dependencies

Usability dependencies state the simple fact that resource users should be able to properly use produced resources. This is a very general requirement that encompasses compatibility issues such as:

- data type compatibility
- format compatibility
- database schema compatibility
- device driver compatibility

The exact meaning and range of usability considerations varies with each kind of resource. Section 4.6, which describes specializations of flow dependencies for a variety of different resources, also discusses in more detail the meaning of usability dependencies for each of them.

4.5.1.2 Managing usability dependencies

One interesting observation resulting from this work is that, irrespective of the particular usability issue being managed, coordination alternatives for managing usability dependencies can be classified using the following two design dimensions (Figure 4-5) :

Design Dimension	*Design Alternatives*
Who is responsible for ensuring usability?	- Designer (Standardization) - Producers - Consumers - Both producers and consumers - Third party
When are usability requirements fixed?	- At design-time - At run-time

Figure 4-5: A framework for managing usability dependencies.

• *Who is responsible for ensuring usability ?*

The following alternatives are possible:

- *Designer is responsible (No run-time coordination is necessary).* Components are specially selected at design-time so as to be compatible with one another. This is often achieved by developing applications using standardized components. Examples of standardized component families include OLE objects, OpenDoc components, Visual Basic VBXs, etc [Adler95]. The advantages of this approach include run-time efficiency and reliability. On the other hand, it limits the choice of components for a particular functional requirement to those explicitly designed for the particular standardized environment.

- *Producers are responsible for ensuring usability.* This implies that the producer knows the format expected by the users, and is able to generate or convert its resources to the user format.

- *Consumers are responsible for ensuring usability.* This requires the consumer to recognize the format of resources it receives, and to be able to convert them to its own format, if necessary.

- *Third party ensures usability between producers and consumers.* Third party must know and be able to handle both formats.

- *Both producers and consumers convert to and from an interchange format.* The advantage of this approach is that it does not require prior knowledge of the formats produced and expected by producers and users. This is particularly desirable if producers and users are dynamically changing, and each of them is using a different native format. The disadvantage is that two conversions take place, which might be

inefficient if conversions are computationally costly. Producers and users must agree on the interchange format.

- *Are usability requirements fixed ?*

Coordination processes can be further classified depending on whether the producer and consumer formats are fixed and known at design-time, or whether they are negotiated at run-time. In the latter case, the management of usability dependencies might introduce additional flow dependencies to the system, that have to be managed in turn.

4.5.2 Accessibility Dependencies

4.5.2.1 Types of accessibility dependencies

Accessibility dependencies specify that a resource must be accessible to a user before it can be used. Since users are software activities, accessibility specifies more accurately that a resource must be accessible to the process that executes a user activity before it can be used. Important parameters in specifying accessibility dependencies are the number of producers, the number of users, and the resource kind.

4.5.2.2 Managing accessibility dependencies

There are two broad alternatives for making resources accessible to their users (Figure 4-6):

- Place producers and users "close together"
- Transport resources from producers to users

Depending on the type of resource being transferred, either or both alternatives might be needed. Placing producer and user activities "close" to one another generally decreases the cost of transporting the resource. Combinations of placing activities and transporting resources should be considered in situations where the cost of placing the activities is lower than the corresponding gain in the cost of transporting the resource.

The following is a discussion of the two alternatives:

Principal design alternatives	*First-level of specialization*	*Second-level of specialization*
Place producers and consumers "close together"	• Place at design-time	- Package in same sequential module - Package in same executable - Assign to same processor - Assign to nearby processors
	• Place at run-time	- Code is accessible to all processors - Physical code transportation required
Transport resource	*Actual processes depend on resource kind (see Section 4.6)*	

Figure 4-6: A framework for managing accessibility dependencies.

a. Place producers and users "close together"

This can be done either at design-time, or at run-time.

- *Place activities at design-time.* The following is a list of ways to manage this step, in decreasing order of efficiency:

  - ♦ *Package activities together in the same sequential code block.* In this case, transport of data resources becomes trivial through the use of local variables. However, such a packaging is subject to a large number of restrictions (all activities must be in source code form; they must be written in the same language and must be assigned to the same processor and executable; also, data resource must be transportable through local variables) and is not always possible.

  - ♦ *Package activities in the same executable.* Transport of data resources can be done cheaply through global variables.

  - ♦ *Assign activities to the same processor.* Transport of data resources can be done through shared memory.

  - ♦ *Assign activities to neighboring processors.* Transport of data resources will require network communication, but still potentially cheaper than if producer and users were randomly assigned.

- *Move activities at run-time.* Either producers can be moved close to users or vice-versa. In the simplest case this would imply assigning the producer to the same processor where user activities are also assigned. In more complicated cases, this step might require physically transferring an activity's code to the target machine.

b. Transport resource from producers to users

This step depends on the kind of resource that is flowing. It is discussed in more detail in Section 4.6.

4.5.3 Prerequisite Dependencies

4.5.3.1 Types of prerequisite dependencies

A fundamental requirement in every resource flow is that a resource must be produced before it can be used. This is captured by including a prerequisite dependency in the decomposition of every flow dependency.

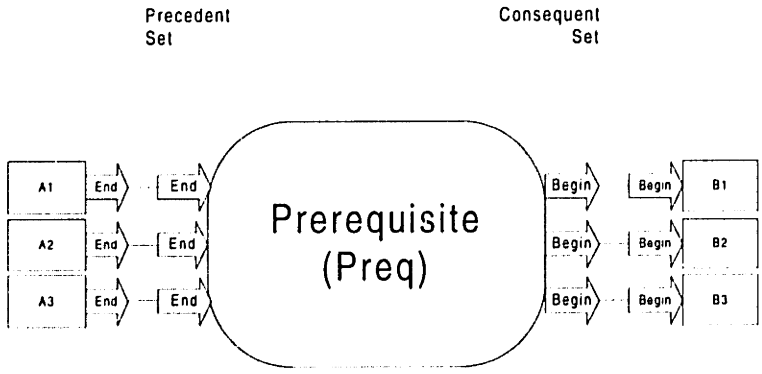


Figure 4-7: Prerequisite dependency.

Prerequisites are relationships between two sets of activities (Figure 4-7). In the following discussion we will refer to set A as the *precedent set* and to set B as the *consequent set.* As is the case with flow dependencies, prerequisite dependencies in our vocabulary have *stream semantics*: they assume that precedent and consequent activities might execute multiple times over the lifetime of an application execution. Prerequisite dependencies thus specify constraints on the allowed execution interleavings of precedent and consequent activities.

Prerequisite dependencies occur very frequently in software architectures. They are the most frequently used member of the dependency family we call timing dependencies. Timing dependencies express constraints in the timing of control flow into a set of activities. They are discussed in Section 4.7.

Prerequisite dependencies form a family of related sequencing constraints. The most useful members of the prerequisite family are the following:

- *Persistent prerequisites* specify that a single occurrence of activity A is an adequate prerequisite for an infinite number of occurrences of activity B. This requirement arises often in system initialization processes: An initialization activity must be executed once before any number of system use activities can take place.

- *Perishable prerequisites* are a special case of permanent prerequisites. They specify that a single occurrence of activity A is an adequate prerequisite for an indefinite number of occurrences of activity B. However, occurrence of a third activity C invalidates the effect of activity A, which must then be repeated (Figure 4-8). Examples of this dependency arise in situations where resources (communication channels, files) are opened and periodically closed. If no activity C is connected to their invalidation port, perishable prerequisite dependencies become identical to persistent prerequisites.

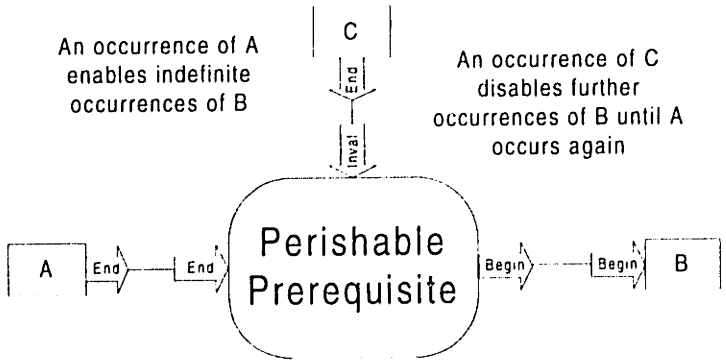


Figure 4-8: Perishable prerequisites.

- *Cumulative prerequisites* permit occurrences of activity A and activity B to be interleaved as long as the number of occurrences of activity B is always smaller than or equal to the number of completed occurrences of activity A. This prerequisite arises in asynchronous resource flows with buffering.

- *Transient prerequisites* specify that at least one new occurrence of activity A must precede each new occurrence of activity B. Transient prerequisites satisfy the definition of cumulative prerequisites and can be thought of as a special case of that dependency type. Perishable prerequisites reduce to transient prerequisites (Figure 4-8), when B and C are the same activity.

- *Lockstep prerequisites* specify that *exactly one* occurrence of activity A must precede each occurrence of activity B. They occur in resource flows without buffering, where it must be ensured that every produced element is used before the next one can be produced. Lockstep prerequisites are a special case of transient prerequisites.

The above variations of prerequisite relationships can be organized in a specialization hierarchy, as shown in Figure 4-9. The implication of prerequisite specialization relationships is that coordination processes for managing a prerequisite relationship can also be used to manage any of its parent relationships in the specialization structure. For example, in order to manage a cumulative prerequisite, in addition to using processes specifically designed for this type of prerequisite, designers can also consider using coordination processes for transient or lockstep prerequisites.

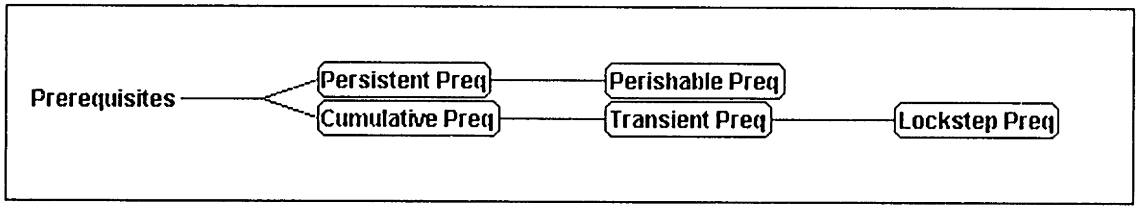


Figure 4-9: Specialization relationships among different prerequisite dependency types.

Prerequisite dependencies can be further classified according to:

- the number of precedent activities
- the number of consequent activities
- the relationship (and/or) among the precedent activities: In *And-prerequisites*, all activities in the precedent set must occur before activities in the consequent set can begin execution. By contrast, in *Or-prerequisites*, occurrence of at least one activity in the precedent set satisfies the prerequisite requirement.

4.5.3.2 Managing prerequisite dependencies

There are four generic processes for managing prerequisite dependencies (Figure 4-10):

a. Producer Push

This process decomposes into a control flow dependency (Section 4.6.1). The alternatives for managing it are the same as those of managing the corresponding control flow dependency.

Producer push processes manage lockstep prerequisites. Consequents are invoked once each time the precedents complete execution.

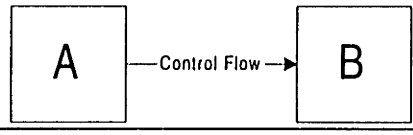
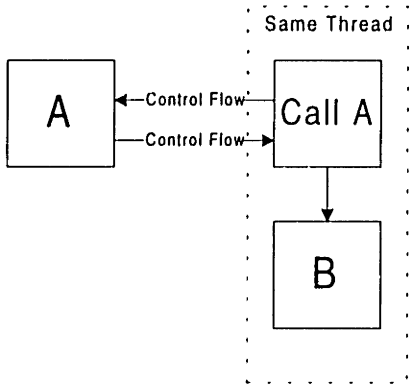
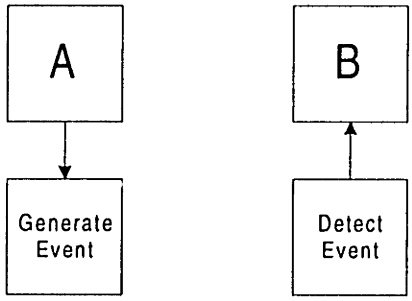
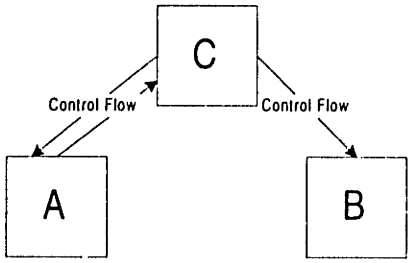
Producer Push		As soon as the precedent completes, it invokes the consequent by explicitly passing control to them.
Consumer Pull		Before it begins execution, the consequent synchronously calls the precedent.
Peer Synchronization		Both precedent and consequent are executed by independent threads of control and synchronize themselves using shared events.
Controlled Hierarchy		A third party controls the invocation of both the precedent and the consequent

Figure 4-10: Generic processes for managing prerequisite dependencies.

b. Consumer Pull

This process family decomposes into a synchronous call pattern of control flow dependencies. The alternatives for managing it are the same as those of managing the corresponding pattern of control flows.

Consumer pull processes manage lockstep prerequisites. Precedents are invoked once before each consequent. Consumer pull organizations can also be used to manage persistent and perishable dependencies: Before starting itself, each consequent checks whether the prerequisite condition is valid, and invokes the precedent activities if it is not.

c. Peer Synchronization

Peer synchronization processes can be used to manage all kinds of prerequisites. Figure 4-11 shows their generic form for each kind of one-to-one prerequisite. All processes can be generalized to handle many-to-many prerequisites.

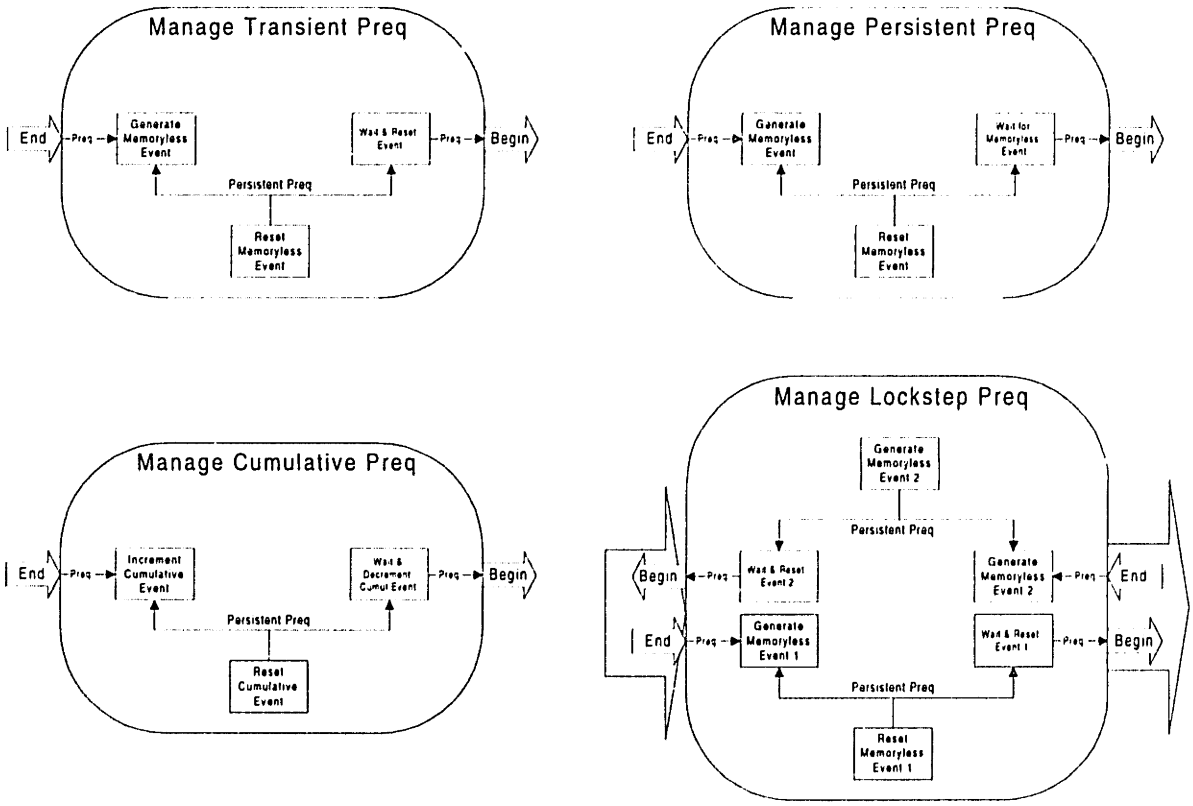


Figure 4-11: Generic processes for managing prerequisite dependencies using peer event synchronization.

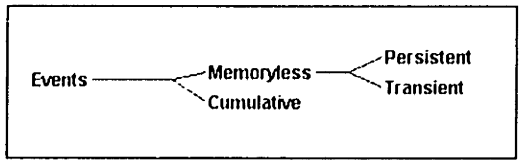
Peer Synchronization processes rely on the generation and detection of shared events in the system. Events can be classified as follows (Figure 4-12):

- *Memoryless Events.* Such events are only capable of recording binary states (occurred/did not occur). They can be used for managing permanent, perishable transient, and lockstep prerequisites.

Memoryless events can be further distinguished into persistent and transient.

Persistent event protocols keep a record of whether the event has occurred or not. Therefore, the detection of the event need not take place at the time of event generation. *Transient* event protocols do not keep a record of whether an event has occurred. Events are either detected at the time they are generated, or go unnoticed. This requires some extra coordination to make sure that event detection activities have been started before event generation takes place.

• *Cumulative Events.* Cumulative events are capable of remembering how many times they have occurred. They are used in the management of cumulative prerequisites. Apart from being resetted, cumulative events can be incremented and decremented.



Persistent Memoryless Events

Event type	*Generate*	*Detect*	*Reset*
Semaphore	Signal Semaphore (V)	Wait on Semaphore (P)	
File Creation	Create File	Test File Existence	Delete File
File Modification	Write File	Compare file modification time with stored modification time	Set stored modification time to file modification time
Process Creation	Create Process	Test Process Existence	Kill Process

Transient Memoryless Events

Event type	*Generate*	*Detect*
UNIX Signal mechanism	signal system call	wait system call
Windows DDE mechanism	send DDE transaction	initialize DDE conversation

Cumulative Events

Event	*Reset*	*Increment*	*Decrement*	*Detect*
Counting Semaphore	Set semaphore to zero	Signal semaphore	Wait on semaphore	
List	Clear List	Add element to list	Remove element from list	Check if list is empty

Figure 4-12: A taxonomy and examples of event types.

d. Controlled Hierarchy

There are three variations on this process:

- third party synchronously calls precedent, then calls consequent.
- third party asynchronously calls precedent, then schedules consequent after sufficient delay
- third party schedules both precedent and consequent with sufficient relative delay

Controlled hierarchy processes can be used to manage lockstep prerequisites. Permanent prerequisites can also be managed by this approach by placing the prerequisite code before any other code in the system (e.g. in an initialization module or at the top of the main program).

Figure 4-13 summarizes the design dimensions of prerequisite dependencies and coordination processes.

Principal Design Dimensions	*Design Alternatives*	*Other Design Dimensions*
Type of prerequisite	- Persistent - Perishable - Cumulative - Transient - Lockstep	
Organization of coordination mechanism	- Producer Push - Consumer Pull - Peer Synchronization - Controlled Hierarchy	- Synchronous vs. Asynchronous control flow - Type of shared event used - Wait for precedent completion vs. pre-scheduling

Figure 4-13: A framework for managing prerequisite dependencies.

4.5.4 Sharing Dependencies

4.5.4.1 Types of sharing dependencies

Sharing arises when more than one activity requires access to the same resource. Sharing dependencies can be specialized using the three dimensions of the resource-in-use framework of Section 4.4.4. For each different combination of resource-in-use parameters

(e.g. indivisible, consumable, concurrent), a different specialization of sharing dependency can be defined.

Sharing dependencies arise in one-to-many, many-to-one, and many-to-many flow dependencies in two distinct situations:

- Resource sharing
- User sharing

a. Resource sharing

Resource sharing considerations arise in one-to-many flow dependencies, because more than one activity uses the same resource. Resource users are assumed to be independent. Therefore, the sharing coordination requirements depend solely on the sharing properties of the resource. The different possibilities are:

• *Divisible resources*	Resources can be divided among the users.
• *Consumable resources*	The total number of users must be restricted.
• *Nonconcurrent resources*	The number of concurrent users must be restricted .

b. Consumer sharing

Consumer sharing dependencies arise in many-to-one flow dependencies because more than one producers produce for the same consumer activity, viewed as a "resource". Consumer "resources" can be characterized using the resource-in-use framework. The different dimensions are:

Divisibility	Consumer activities either occur or do not occur. Therefore, they are considered indivisible resources.
Consumability	Consumability of a consumer activity means that it can occur a limited number of times or, equivalently, that it can accept a limited number of produced resources. This implies the need for coordination in order to select which resources will be accepted. Modeling consumer activities as consumable resources enables many-to-one flow dependencies to be used for modeling *race conditions.*
Concurrency	Concurrency determines whether multiple instances of a consumer activity can be active at the same time. Some consumer activities are nonconcurrent (e.g. non-reentrant procedures). In that case, coordination should be installed to restrict simultaneous execution of more than one activity instances.

Many-to-many flow dependencies contain a combination of both resource and consumer sharing dependencies.

4.5.4.2 Managing sharing dependencies

The problem of resource sharing has been studied extensively by researchers in various areas and there exists a huge literature of related algorithms and techniques. Our purpose in this section is to take an architectural look at resource sharing techniques, showing how their interfaces can be abstracted to a small number of generic processes, and how they relate to the other components of a resource flow management process in a small, well-defined number of ways.

There are three general techniques for coordinating resource sharing requirements (Figure 4-14):

- Divide resource
- Restrict access to resource
- Replicate resource

Resource Type	*Sharing Coordination Required*	*Specializations*
Divisible Resources	• Divide Resource	- Divide before transportation - Divide after transportation
Indivisible Resources		
Consumable and/or Finitely Concurrent	• Restrict access to resource • Replicate resource	- Restrict consumer activity execution - Restrict resource transportation - Restrict resource production
Nonconsumable and Infinitely Concurrent	No sharing coordination is required	

Figure 4-14: A framework for managing sharing dependencies.

a. Divide resource

This technique applies to divisible resources. It can be represented by a process that uses the entire resource and produces a set of new subresources (Figure 4-15). Subresources are considered independent resources and can then flow to each user with no further coordination.

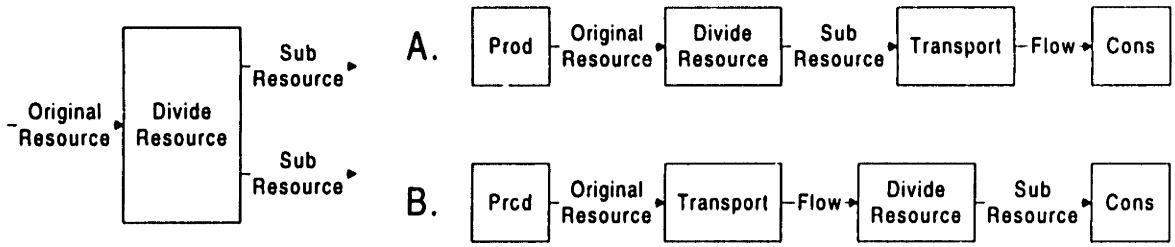


Figure 4-15: Sharing of divisible resources in a flow dependency.

There are two different ways a resource divide can be combined with the rest of a flow coordination process:

- *Divide resource before transportation* (Figure 4-15A). This is the most common case. The entire resource is divided at the site of production, and the generated subresources are independently transported to their users.

- *Divide resource after transportation* (Figure 4-15B). The resource is first transported at each site and a new subresource is extracted locally. Examples of such resources are circulating tokens, in which successive users write or reserve data areas.

b. Restrict access to resource

This very general technique applies to both consumable and nonconcurrent resources. In both cases the function of the coordination process is to restrict the flow of control into activities accessing the resource (Figure 4-16). More specifically:

- For consumable resources, the process restricts the total number of resource accesses.

- For nonconcurrent resources the process limits the total number of concurrent resource accesses. The most common case is when only one concurrent access can be allowed. Then, resource sharing becomes equal to a mutual exclusion dependency [Raynal86].

From an architectural perspective, there are three different ways an access restriction process can be integrated with the rest of a flow coordination process:

- *Restrict consumer activity execution* (Figure 4-16A). This method is used when a resource is accessible to all consumers (e.g. a fixed hardware resource), or when each consumer is using a local protocol to restrict access.

- *Restrict resource accessibility* (Figure 4-16B). This method prevents the resource from being transported to consumers until they are allowed to use it. It has efficiency

advantages in situations where resource transportation is costly (e.g. for large files), and only a subset of the candidate consumers is allowed to use the resource.

- *Restrict resource production* (Figure 4-16C). This alternative should be considered when managing user sharing dependencies. In situations where only a subset of the produced resources is ever used, it might be more efficient to not produce unless usage has been guaranteed.

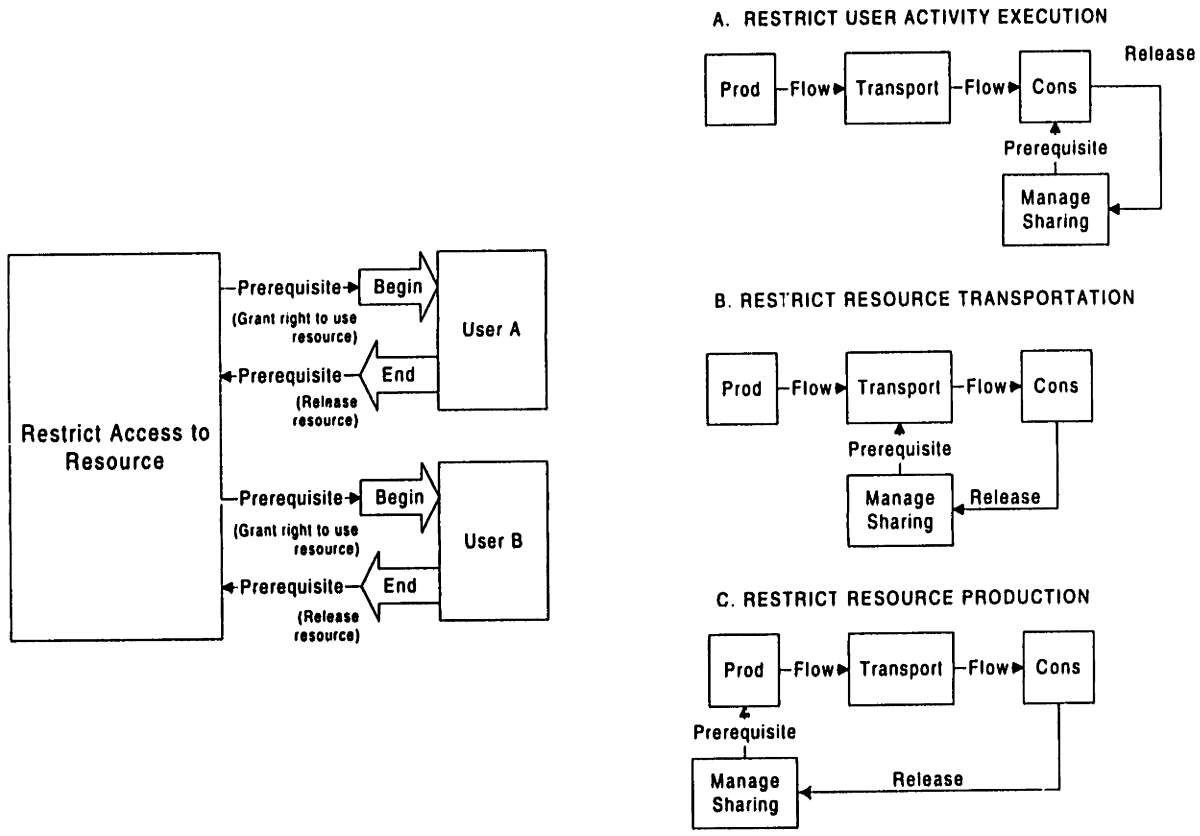


Figure 4-16: Sharing by restricting access to resource.

c. Replicate Resource

Resource replication is a technique that jointly manages accessibility and resource sharing dependencies. Its more general architectural form is similar to that of a resource division process. However, it applies to indivisible resources.

Combinations of division and restriction

The previous techniques can be combined to handle more complex resource sharing requirements. For example, in order to share a resource that is nonconcurrent and finitely

divisible among a potentially infinite number of users a combination of division and access restriction can be used: Whenever an access is desired, extraction of a new subresource is first attempted. If that fails, time-sharing is used. This algorithm is used, for example, to manage the sharing of finite capacity buffered input/output channels among a potentially infinite number of user processes.

4.5.5 Putting it all together: Flow dependencies

The design dimensions of generalized resource dependencies are the sum of the design dimensions of their component dependencies (Figure 4-17). For each combination of dimension values, a different special case of resource flow can be defined. The following is a discussion of the different dimensions and the alternative flow dependencies they can be used to define.

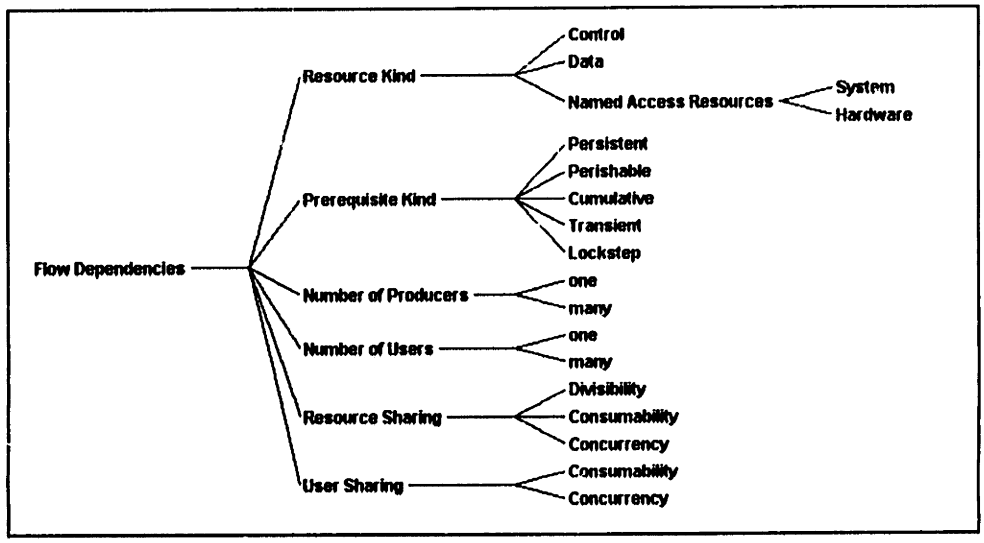


Figure 4-17: First two levels of design dimensions for flow dependencies.

a. Resource kind

The most important design dimension is the kind of resource. Section 4.6 will describe how resource dependencies are specialized according to this dimension.

b. Prerequisite relationship

Another important dimension is the kind of prerequisite requirement. According to this dimension, resource relationships are classified into:

- *Persistent Flows.* Such dependencies describe situations where one or more resources are produced once, and can then be used an infinite amount of times. In software

systems, they arise often to describe the use of calculated constants, and system resources (printers, network channels, etc.) which are set up once and then used an indefinite amount of times.

- *Perishable Flows* are a more refined special case of permanent dependencies. They describe situations where produced resources can be used an indefinite amount of times until they become invalidated. File caching provides an example application where this type of flow describes the underlying interdependency: Cached file blocks are transferred (produced) once from disk, and can then be read an arbitrary number of times until some other process modifies their corresponding disk block. In that case, cached file blocks become invalidated and have to be refreshed from disk before they can be read again.

- *Cumulative Flows* describe situations where every resource produced can only be used once. Producers and users can proceed asynchronously, but at no time can the number of user accesses exceed the number of produced resources. Reading and writing a pipe channel by two separate processes is an example of this type of flow.

- *Transient Flows* describe situations where a stream of resources is produced, but the use of each resource is optional. Thus, new resources in the stream can overwrite previous ones, possibly before they have been used. One example application where transient flows describe the underlying interdependency is a log file that is periodically being updated and can be printed by a user at will. Not all versions of the file need to be printed. Therefore, new updates can overwrite the previous contents of the file without the need for additional coordination.

- *Lockstep Flows* describe situations where there must exist tight synchronization between producers and users of resources. All resources produced must be used and no resource can be produced until all previous resources have been used by all designated users. Stream data flows using non-buffered (indivisible) carriers are examples of lockstep flows.

Managing Prerequisite Coordination Processes

The coordination process selected to manage a prerequisite dependency at the heart of a resource flow has a profound influence on the overall organization of the interacting activities. Corresponding to the four generic classes of prerequisite coordination processes we have an equivalent taxonomy of flow organizations:

- *Producer Push.* In push organizations, also called *eager flows*, resource users explicitly receive control from producers every time a new resource has been produced. Only lockstep flows can be implemented in this manner.

If control is transferred to users using synchronous calls, this organization reduces to what is commonly called *client/server architecture*. In this case, resource producers acts as clients and resource users act as servers.

- *Consumer Pull.* In pull organizations, also called *lazy flows*, resource producers are invoked by users whenever the latter require a new resource. Only lockstep and permanent flows can be implemented in this manner.

  Pull organizations of flow dependencies also reduce to *client/server architectures*. In this case, resource users act as the clients and resource producers act as the servers. Note, however, that the direction of the client/server relationship is the *inverse* of the direction of the flow relationship.

- *Peer synchronization.* In peer organizations producers and users are executed by separate threads of control and synchronize themselves through events. This is a more loose organization, appropriate for managing all kinds of flows. It is particularly suitable for organizing cumulative and transient flows. It might not be as efficient for managing lockstep flows, where tight synchronization is required. Examples of flow processes that are organized in this manner include pipe channel flows, shared memory flows with separate semaphore synchronization, tuple space flows, etc. Ada's *rendezvous* interprocess communication paradigm [DoD83] is one well-known specialization of peer organizations. Other researchers have used the term *implicit invocation architectures* to characterize such organizations [Garlan88].

- *Controlled hierarchy.* Such organizations typically result in systems with centralized control, where a main program explicitly controls the sequence of flow participants.

c. Number of producers and users - Sharing dimensions

- *One-to-one dependencies.* These are the simplest kinds of dependencies. The defining dimensions are the kind of resource and the kind of prerequisite relationship. There are no sharing considerations.

- *One-to-many dependencies.* In one-to-many dependencies, resource sharing becomes an issue. Different dependency types can be defined for each combination of resource sharing dimensions of each of the users. Some interesting special cases are:

one-to-all	Each resource flows to all users
one-to-one-of-many	Each resource flows to one of the users. This can be managed in an application-independent way (e.g. first come-first served), or in an application-specific way. In the latter case, user consumer ports usually provide additional pieces of information, such as user priorities.

- *Many-to-one dependencies.* In many-to-one dependencies, user sharing issues have to be addressed. Users might not be willing to receive all resources produced, or they might not be able to receive them concurrently.

  Situations where users are not willing to receive all resources produced are often referred to as *race conditions*. In our framework, race conditions are modeled as many-to-one dependencies where the user activity acts as a consumable "resource". General ways of managing consumable resources can be used to manage the dependency.

- *Many-to-many dependencies.* These dependencies are the most complex family, because they can specialized according to both resource and user sharing dimensions. Some interesting special cases include:

each-to-all	Every resource produced flows to all users
each-to-one	Every resource produced flows to one user only
each-from-one	Each user receives one resource only
all-from-one	Only one of the resources produced flows to (all) users

The design alternatives for managing resource dependencies are the product of the different alternatives for managing each component dependency. In principle, each of the component dependencies can be managed by independent coordination processes. In practice however, there often exist opportunities to increase efficiency by managing patterns of dependencies using joint coordination processes. This gives rise to additional design alternatives that designers should be aware of. We have already encountered the opportunity to use *joint* coordination processes for managing accessibility and sharing (restrict resource transportation, replicate resource). In the following sections, we will encounter more opportunities for joint dependency management.

4.6 Special Cases of Flow Dependencies

This section will demonstrate how the generic flow coordination model of the previous section can be specialized and form the basis for a large number of different practical mechanisms for managing flow dependencies.

We distinguish flow dependencies according to the kind of resource that flows. For each kind of resource, we begin by discussing how the components of the model of the previous section specialize to handle issues specific to that kind of resource. Just as we did in the previous section, we then show how a design space for the total dependency can be defined by combining the components of the model. We conclude each section with a number of concrete example coordination processes, showing how they can be derived from the generic coordination model.

4.6.1 Control Flows

Control flows specify relationships of the resource type commonly referred to in Computer Science as *control*. From a resource perspective, control is more accurately described *as a thread of processor attention*. Therefore, control flows specify how threads of processor attention must flow from one set of activities to another.

Every software activity needs to receive control *from somewhere* in order to begin execution. In SYNOPSIS descriptions, control flows into the Begin port of an activity, causing it to begin execution. It flows out of its End port after it has completed execution (see Section 3.3.3). From a resource perspective, each activity consumes a thread of control in order to begin execution, and produces a new thread of control upon completion.

Control flow dependencies do not generally appear at the top-level of software architecture descriptions. They are either left implicit, in which case they are automatically added and managed by the SYNTHESIS system (see Section 5.2.4), or they are introduced during the management of timing dependencies, such as prerequisites (Section 4.5.3.2). Since prerequisite dependencies are part of every flow dependency, control flow dependencies end up occurring in coordination processes of almost every dependency.

4.6.1.1 Specialization of the generic process

The following table summarizes the most important differences in the management of control flows from the generic case.

- Programming languages and operating systems support primitive mechanisms for transferring control among activities.
- Prerequisites are automatically managed.
- Control transfer to multiple activities is always done by replication.

a. Usability.

Control corresponds to a thread of processor attention in a given run-time environment. Consumption of control by a software activity corresponds to enactment of that activity in the respective run-time environment. For a software activity to properly use control, it must be compatible with its run-time environment.

This dependency is generally managed at design-time, by properly packaging, compiling or configuring activities for their intended run-time environments. If activities are already in executable form, they must be placed in compatible execution environments and control must be transported to those environments (possibly by the use of remote calls).

In some cases, limited run-time configuration is also possible, usually by setting environment variables before passing control to the activity. For example, an environment variable can be set before passing control to a numerical analysis program to specify whether a math co-processor exists in the run-time system or not.

b. Accessibility.

Accessibility dependencies specify the fact that a software activity must be accessible to a processor in order to receive control from it (be enacted by it). For software activities, accessibility implies accessibility of the program text.

The generic accessibility coordination alternatives presented in Section 4.5.2.1 apply here as well. As in the generic case, placing resource producers and users close together at design time, reduces the effort of actually transporting control between them.

Transport control from producer to users

Transportation of control relies in a number of low-level programming language and operating system mechanisms that are modeled in our framework as primitive coordination processes (Figure 4-18). The most common such mechanisms are presented below:

- *Sequentialization* (Seq). This is the most basic control flow mechanism. It manages the flow of control from activity A to another activity B by placing A before B in the same sequential code block. It is a design-time coordination process in that, all necessary work (ordering and packaging of activities) takes place before run-time. It can, of course, only be used if both A and B can be packaged in the same code block. This implies that they must both be written in the same language, and assigned to the same executable and processor.

  Some operating systems support variations of Seq that ease the above restrictions. For example, UNIX supports the exec system call, using which control can be passed (without return) to a different executable program.

  In its most general form, Seq can manage a one-to-many control flow relationship. The semantics of a one-to-many Seq is that the follower activities are placed after the source activity in the same sequential code block. Ordering among the follower activities is left unspecified.

Dependency Type	*Generic Mechanism*	*Examples*
one-to-one	*any of the following mechanisms*	
one-to-many	• Sequentialization	
many-to-one	• Asynchronous Call	- UNIX `fork` - Multilisp `future`
	• Synchronous Call	- Procedure call - RPC
	• Scheduling	- UNIX `cron`
many-to-many	• Broadcasting	- ISIS Multicast [Birman89]

Figure 4-18: Primitive control flow mechanisms.

- Asynchronous Call (Fork). This mechanism manages a many-to-one control flow. It generates a new thread of control and enables the enactment of an activity from a variety of other activities. There is a rich variety of such mechanisms that enable asynchronous creation of new local or remote threads, based on the same or on different executable programs. Figure 4-18 shows some variations of Fork supported by different systems.

- *Scheduling.* This mechanism also manages a many-to-one control flow. However, the new thread of control gets created after a specified delay. Most operating systems provide such a mechanism, based on the system timer. When delay is set to zero, scheduling becomes equivalent to an asynchronous call. In fact, in some systems (e.g. Visual Basic), this is the only way to implement asynchronous calls.

- *Synchronous Call* (Call). This is the most widely used mechanism to pass control in today's programming languages. Most languages and systems support some variation of this mechanism, ranging from simple procedure calls, to complex RPC with elaborate semantics. In our control flow dependency framework, it manages a composite pattern of two opposite direction control flows. If the return control flow is ignored, synchronous calls can also be used to manage many-to-one control flows.

- *Broadcast.* This process manages many-to-many control flows. It is syntactically similar to an asynchronous procedure call, in that it enables a named entity to be invoked from a variety of places in a program. However, in this case the named entity corresponds to a group of activities, and the broadcast creates a set of new threads that cause simultaneous execution of all activities in the group. Few systems provide built-in support for broadcast primitives.

c. Prerequisite

Prerequisite dependencies specify the requirement that a resource is produced before it can be used. Since an activity cannot start before a thread of control has been generated and passed to it, in the special case of control flows this requirement is always automatically managed.

d. Sharing.

- *Control Sharing.* From the point of view of the resource in use framework (Section 4.4.4), control is:

  - *Consumable.* A thread of control is completely "used up" to start a single activity. Upon termination, this activity will produce a new thread of control.

  - *Nonconcurrent.* This is a corollary of the consumability property.

  - *Indivisible.* A thread of control cannot be divided into more than one activities. However, new threads of control can easily be generated to handle additional activities.

  Sharing of control flow arises in one-to-many dependencies. The consumability of control implies that each thread of control can only be used by a single activity. In order to manage one-to-many dependencies, control replication, that is, explicit generation of new control threads must take place for each user.

- *User Sharing.* The general issues of user sharing described in Section 4.5.4 apply in this case.

4.6.1.2 Managing control flow dependencies

In the special case of control flows, the only flow design dimensions (see Figure 4-17) that matter are the number of activities at each side of the flow and (for many-to-one flows) any user sharing restrictions.

a. One-to-one flows

One-to-one control flows can be managed by specializing any of the built-in one-to-many, many-to-one, and many-to-many primitive mechanisms presented in the previous section. The particular form and location of the two interdependent components will determine which mechanisms are applicable.

b. One-to-many flows

One-to-many control flows can be managed

- directly by one-to-many primitive mechanisms, such as sequentialization
- by specializing many-to-many primitive mechanisms, such as broadcast
- by decomposition into equivalent sets of one-to-one flows

c. Many-to-one flows

Many-to-one control flows are usually managed by variants of the call mechanism (asynchronous and synchronous). Additional user sharing coordination might be required to handle special situations, such as non-reentrant control recipients or race conditions.

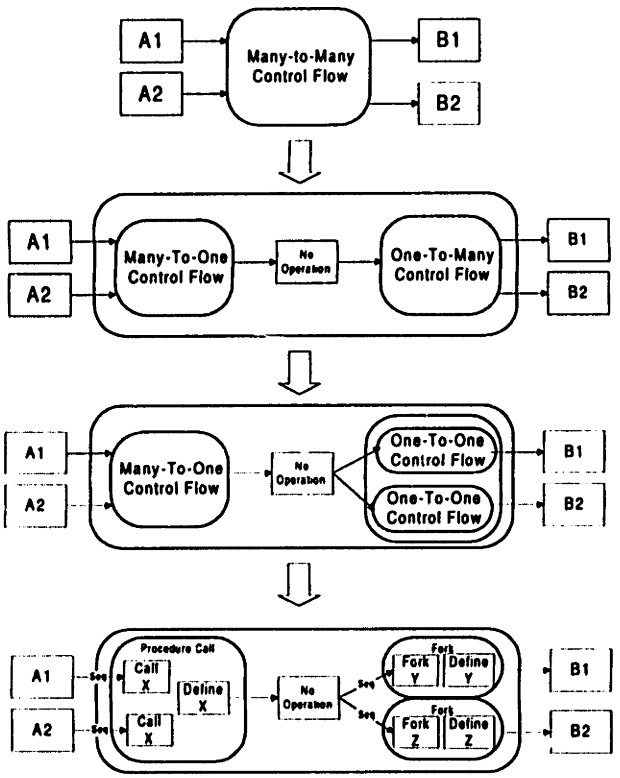
d. Many-to-many flows

Depending on their exact semantics, many-to-many control flows are managed by many-to-many primitive mechanisms, or by decomposing them into simpler patterns of control flow

---

Example 6-1: Managing an each-to-all control flow.

We are interesting in managing a many-to-many control flow dependency in a system that does not support broadcast primitives. One approach is to decompose the flow into an equivalent pattern of simpler control flows. Many-to-one flows can be managed by a local procedure call, while one-to-many calls can be further decomposed into one-to-one calls. Finally, one-to-one flows can be managed by a variety of methods, such as asynchronous calls (Fork). Figure 4-19 shows the successive transformations and resulting code fragments.




```
Thread of activity A1:
...
A1
call X
...

Thread of activity A2:
...
A2
call X
...

procedure X
  fork Y
  fork Z
end procedure

procedure Y
  B1
  ...
end procedure

procedure Z
  B2
  ...
end procedure
```

Figure 4-19: Managing a many-to-many control flow by decomposition into simpler dependencies.

4.6.2 Data Flows

Data flows specify relationships between producers and users of data values. In data flows, producer ports decompose to data out ports, while consumer ports decompose to data in ports.

4.6.2.1 Specialization of the generic model

The following table summarizes the most important differences in the management of data flows from the generic case.

- Usability management specializes to data type and format compatibility management.
- Data transportation from producer to users becomes the principal design issue.
- Programming languages and operating systems support primitive mechanisms for joint management of data transportation and prerequisite dependencies.

a. Usability

Usability dependencies specify that resource users should be able to properly use produced resources. For data resources, this usually translates to data type and format compatibility issues.

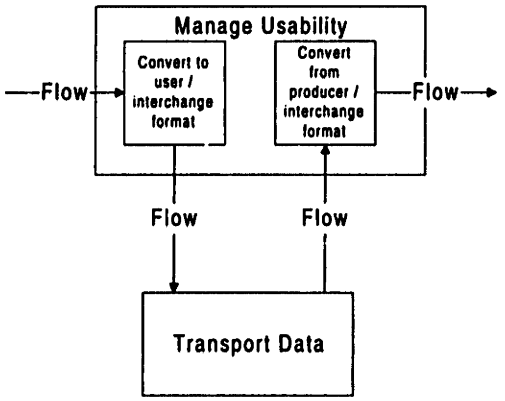


Figure 4-20: Generic process for managing usability in data flows.

The most generic process for managing data usability dependencies is shown in Figure 4-20. It can be specialized to cover all five alternatives outlined in Section 4.5.1. Although its generic form is simple, its implementation depends on the development of a sufficiently rich family of data type and format conversion activities. For simple data resources, such conversions are usually straightforward.

b. Accessibility

The generic accessibility coordination alternatives presented in Section 4.5.2.1 apply here as well. As in the generic case, placing resource producers and users close together at design time, reduces the effort of actually transporting data resources between them.

Transport data from producer to users

There are two broad alternatives here:

- Manage at design-time
- Use carrier-resource at run-time

a. *Manage at design-time*

This mechanism is applicable for transporting preexisting, constant data resources among activities. Each activity simply receives a local copy of the constant at design-time and no coordination takes place at run-time.

b. *Use carrier resource at run-time*

This is the most frequently used method for transporting data. A *repository* or *carrier resource*, accessible to both producers and users, is created. Producers write the data to the shared carrier. Users read the data from the shared carrier. A prerequisite dependency specifies the requirement that carrier resources can be read only after they have been written (Figure 4-21).

Examples of carrier resources are local variables, global variables, shared memory, files, pipes, network channels, tuple space, etc.

Note that this process creates a distinction between the original data resource that is being transported and the carrier resource that transports it. For example, in a string transport process using shared files, it distinguishes between a string resource that is being produced and consumed, and the file resource that transports it from producers to users.

The introduction of a new resource into the system results in the introduction of an additional resource dependency. In contrast to simple data flow dependencies, where data itself is being transported from producers to users, the new resource dependency communicates a resource identifier from producers to users. It is an instance of a *named resource dependency*, that will be described in the next section.

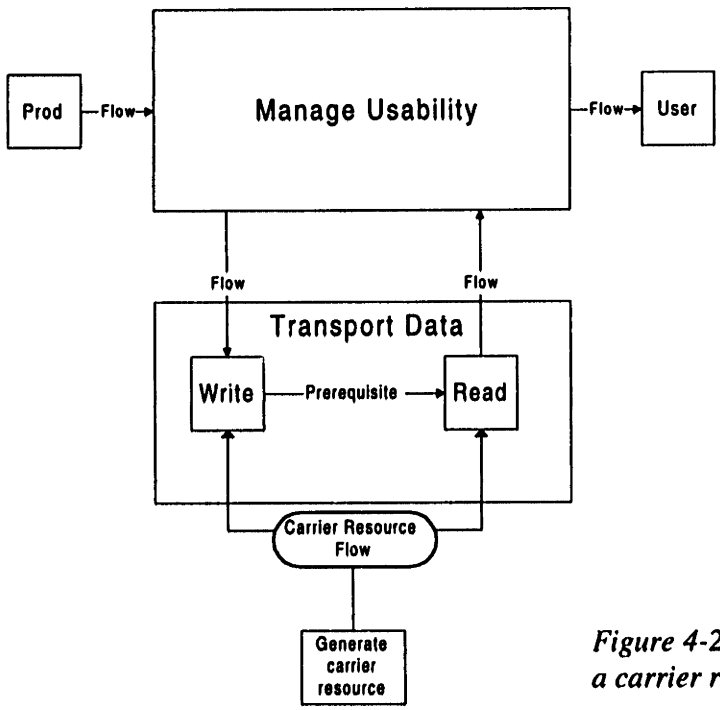


Figure 4-21: Data transportation using a carrier resource.

Carrier resources are classified and selected according to the following two dimensions:

- Their sharing properties
- Their dependence on the run-time locations of their endpoints

i. Sharing properties

Carrier resources are being shared among all writers and readers of a transported data resource. In addition, since the default semantics of data flow imply the flow of a *stream* of data values over the lifetime of an application (see Section 4.5), carrier resources are also shared among different write-read transactions which might be concurrently in progress. Carrier resources selected to manage a particular dependency must thus be able to support the corresponding pattern of resource sharing. Using the resource-in-use framework of Section 4.4.4, we can connect carrier resource sharing properties to their usability in specific data flow dependency patterns:

- *Consumability* of a carrier resource determines whether it can be used in one-to-many flows. In order to be usable in one-to-many flows, carrier resources must be able to support nonconsumable (nondestructive) read operations. Examples of resources that do are variables and files. Examples of resources that do not are pipes and channels.

- *Divisibility* of a carrier resource determines whether it can be used in cumulative stream flows where more than one unread data resources might coexist in the carrier. Indivisible carrier resources cannot be used in cumulative stream flows. Examples of divisible carrier resources are buffered channels, such as pipes, and associative repositories, such as tuple space. Examples of indivisible carrier resources are scalar variables.

- *Concurrency* of a carrier resource determines whether the data consumers can access the data concurrently.

ii. Dependence on run-time location of endpoints (Endpoint location dependence)

Some carrier resources can be generated independently of the run-time location of their users. An example are files in a universally shared file system. Their pathname is universally recognized and therefore can be chosen independently of where the user activities are located at run-time. Other carrier resources are dependent on the run-time location of their endpoints. Example of the latter class are private network channels set up between two dynamically specified endpoints.

From a coordination process perspective, the latter class of resources requires additional flows from the users to the carrier producer activity. The flows must communicate information about the location of user processes. This requirement might complicate the management of the flow process.

The following table applies the above framework to a number of frequently used carrier resources.

Carrier resource	*Divisible*	*Consumable*	*Concurrent*	*Endpoint location dependent*
Local variable	No	No	Yes	No
Shared memory	Yes	No	Yes	No
Pipe	Yes	Yes	No	No
File	Depends	No	Depends	No
Socket	Yes	Yes	No	Yes
Tuple space	Yes	No	Yes	No

c. Prerequisite

The prerequisite requirement of generic flow dependencies is replaced by a prerequisite between the write and read activities of the data transport process (Figure 4-21). In accordance with the generic flow model, that prerequisite determines the type of flow. As described in the previous section, carrier resources must be able to support the write-read access patterns specified by the chosen prerequisite. For example, a cumulative prerequisite would define a cumulative flow, that is, a flow where more than one unread data items might coexist at any given time. This would require the selection of a divisible carrier resource, such as a buffered queue.

The alternative prerequisite types and coordination processes discussed in Section 4.5.3 apply here as well.

Simultaneous management of Transport and Prerequisite

Programming languages and operating systems support a number of built-in mechanisms that simultaneously manage data transport and prerequisite dependencies. Therefore, an important design choice when managing data flow dependencies is whether prerequisite and data transport requirements will be jointly or separately managed.

Most mechanisms are closely related to the primitive control flow transport mechanisms described in Section 4.6.1.2. They are summarized in Figure 4-22.

Dependency Type	*Generic Mechanism*	*Examples*
one-to-one	• Point-to-point channels	- OCCAM channels [Inmos84] - UNIX sockets
	• Pipes	- UNIX pipes
one-to-many	• Broadcast Calls	-ISIS Multicast [Birman89]
many-to-one	• Asynchronous Calls	- ABCL/1 async message passing [Yonezawa86]
	• Synchronous Calls	- Procedure calls - RPC - MS Windows DDE
many-to-many	• Broadcast Calls	- ISIS Multicast [Birman89]

Figure 4-22: Primitive data transport mechanisms.

We observe that there is an asymmetry between support for many-to-one and one-to-many relationships. While most languages and systems support a variation of procedure call (manages many-to-one relationships), very few systems provide primitive support for one-to-many relationships. As a consequence, while most many-to-one data flows are managed using eager flows (control flows from producer to user), most one-to-many relationships are managed, either using lazy flows (control flows from user to producer and back), or by loose peer organizations, where synchronization takes place using shared events.

d. Sharing

Data flow dependencies assume that all users are independent. Abstract data is inherently nonconsumable and concurrently sharable. However, the sharing properties of the carrier resource might limit the consumability or concurrency of the entire flow. Apart from this observation, the generic design dimensions and design alternatives of sharing dependencies presented in Section 4.5.4 apply to data flows as well.

4.6.2.2 Managing data flow dependencies

The distinctive feature of managing data flow dependencies relative to the generic case lies in the management of the data transport process. The main choice here is whether a primitive joint coordination mechanism for prerequisite and transport will be used, or whether the two dependencies will be managed separately. In the latter case, the two main design dimensions involve the selection of the carrier resource type and the process for managing the prerequisite.

a. One-to-one flows

One-to-one flows can be managed by using one-to-one primitive data flow mechanisms (such as pipes or point-to-point channels) or by specializing any of the coordination processes for managing one-to-many, many-to-one, or many-to-many flows.

b. One-to-many flows

There are few built-in primitive mechanisms for direct management of one-to-many data flows. There are three broad alternative design paths:

- specialize many-to-many primitive mechanisms (such as broadcast)
- separately manage prerequisite and transport. One-to-many flows require the use of carrier resources that allow nondestructive read.
- use lazy flow organizations in which consumers call producers when they are ready to use a new data resource. This transforms the control relationship to a many-to-one in the opposite direction.

c. Many-to-one flows

Many-to-one data flow dependencies are most naturally implemented using the procedure call mechanism (and variants thereof).

d. Many-to-many flows

The implementation of many-to-many flows varies depending on their particular type.

Example 6-2: Managing a one-to-many flow with a race condition

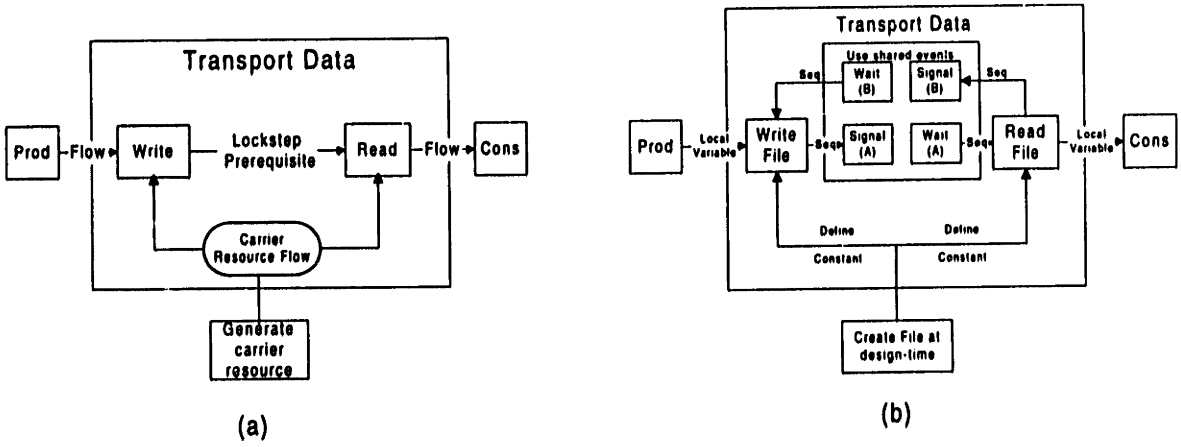


Figure 4-23: Managing a one-to-one-of-many data flow.

We would like to design a coordination process for managing a one-to-one-of-many dependency, that is, a dependency where a stream of data flows from a single producer to multiple potential user activities. However, each data value can only be read by one user.

First, we have to determine the type of dependency that expresses our constraint. It is easy to see that a one-to-many lockstep flow dependency specifies the desirable coordination properties. Assuming there are no usability considerations, the generic decomposition of a coordination process for managing this type of dependency is shown in Figure 4-23(a). The two main parameters in this model are the choice of a carrier resource, and the mechanism for managing the prerequisite. We choose a file as a carrier resource. The filename is selected at design-time and made known, through constant definitions, to all interested parties. We also choose a peer organization for managing the lockstep prerequisite, based on two semaphores[1] (see Figure 4-11). Semaphore A notifies users whenever a new data value has been produced. Semaphore B notifies producers whenever a data value has been used (Figure 4-23(b)). The resulting program fragments at the producer and user threads would look as follows:

```
Producer thread:

...
s = Produce_Value
Wait(B)
Write_File("FILENAME", s)
Signal(A)
...
```

```
User thread:

...
Wait(A)
t = Read_File("FILENAME")
Signal(B)
Use_Value(t)
...
```

Example 6-3: Managing a many-to-one data flow with a non-reentrant user

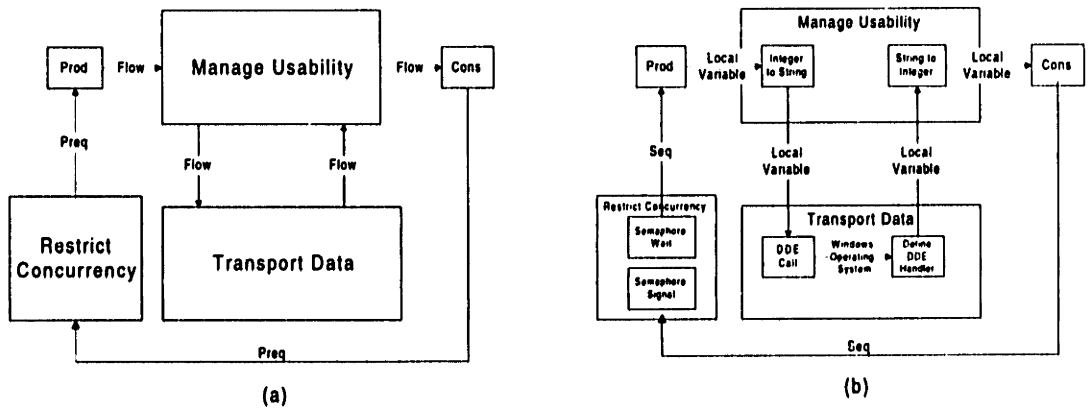


Figure 4-24: Managing a many-to-one data flow with a non-reentrant user.

[1] Initialization steps are omitted for clarity.

We would like to design a coordination process for managing a dependency in which several activities will be sending integer data to a user activity. The user activity is non-reentrant, that is, at most one instance of the user can be executing at any one time.

The user limitations require the use of user sharing coordination that will restrict the concurrency of user invocations. One way of restricting user invocations is by restricting the concurrent production of data (see Section 4.5.4.2). This gives rise to the generic coordination process model of Figure 4-24(a). We have chosen to manage the many-to-one data transport dependency using a Microsoft Windows synchronous call mechanism called Dynamic Data Exchange (DDE). DDE can only be used to transport string data. Therefore, conversion of integers to and from string format is also required. The concurrency restriction (mutual exclusion) part is managed using a semaphore protocol. The resulting code fragments at the producer and user threads would look as follows:

```
Producer Thread:

...
Wait(sema)
n = Produce_Value
s = String(n)
DDE_Call(DDEhandler, s)
...
```

```
User Thread:

procedure DDEhandler(s)
  n = Integer(s)
  Use_Value(n)
  Signal(sema)
end procedure
```

4.6.3 Named Resource Flows

Control and data are examples of direct access resources, that is, resources that are communicated directly from producer to user activities. More complex resource types, such as system or hardware resources, are communicated using a secondary data resource called the *resource name* or *identifier*. We refer collectively to such resource flows as *named resource flows*.

Identifiers allow resources of arbitrary complexity to be accessed through software component interfaces, which typically only support the exchange of relatively simple data values.

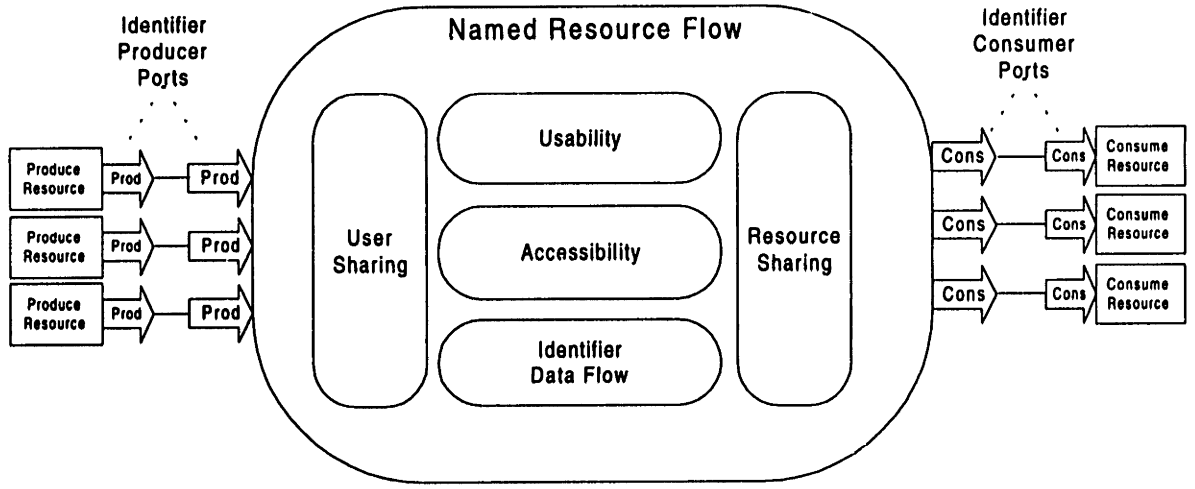


Figure 4-25: Generic model for managing named resource flows.

Figure 4-25 shows the generic form of a named resource flow process. The principal difference of this diagram from the generic process of Figure 4-4 is that the prerequisite dependency of the original diagram has been replaced with an identifier data flow dependency (which, of course, contains a prerequisite). This reflects the fact that, what is actually flowing (i.e. is being exchanged through component interfaces) in such cases is not the resource itself, but its identifier. The general discussion of design dimensions and coordination alternatives of Section 4.5 applies to identifier flows, as well as to the remaining three dependencies of Figure 4-25.

Example 6-4: Coordinating the use of a garbage collectible memory heap

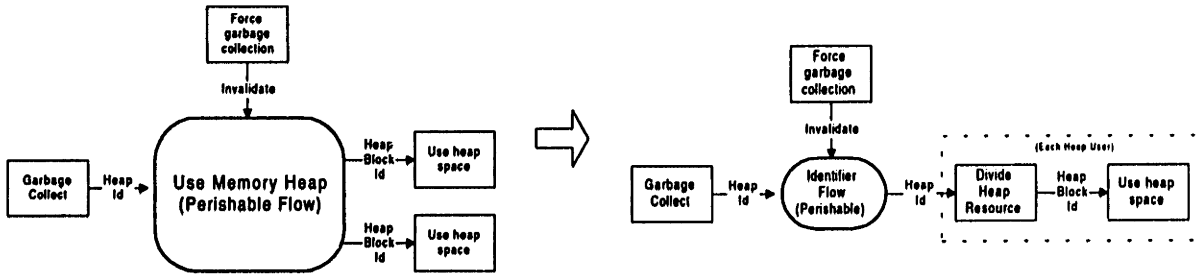


Figure 4-26: The use of a memory heap with periodic garbage collections can be modeled and coordinated using named flow dependencies.

Figure 4-26 shows how the use of a memory heap resource can be coordinated using named flows. In this diagram, a garbage collection process "generates" a new heap resource every time it is called. This resource is then shared among a number of user programs. Periodically, an independent agent forces the system to perform garbage collection, and thus "create" a new heap resource. The situation is naturally described using a perishable flow dependency (see Figure 4-8).

Assuming the heap is accessible to all users, the two issues that have to be resolved in order to manage this dependency are:

- sharing the heap among its users
- managing the heap identifier flow

a. Sharing the heap. Memory heaps are divisible resources (see Figure 4-15). Therefore, the sharing process simply uses the global heap id, reserves a private block, and produces the identifier of that block.

b. Managing the heap identifier flow. The heap identifier flow is a perishable data flow: Each heap id can be used an arbitrary number of times until it is invalidated. Then, a new heap id must be generated (by the garbage collector) before further use can occur. Managing the perishable data flow in requires (see Section 4.6.2.2):

- selecting a carrier resource
- managing a perishable prerequisite

In this case, we store heap identifiers in a global variable, accessible to all users. We manage the prerequisite using a customer pull organization: A separate marker is used to determine whether the current heap id is valid or not. Activity Force Garbage Collection invalidates that marker. Before each access, each heap user checks the marker to see if the heap id is valid. If it is not, it invokes the garbage collector and sets the marker to true.

The resulting code fragments that implement the above pattern of coordination are as follows:

Heap User Thread:

```
...
if not(heap_id_valid) then
   heap_id = Garbage_Collection
   heap_id_valid = True
end if
block_id = Divide_Heap(heap_id)
Use_Heap(block_id)
...
```

Force Garbage Collection thread:

```
...
Force_Garbage_Collection
heap_id_valid = False
...
```

4.6.4 Existing Resource Dependencies

Previous sections have described flow patterns involving resources that are dynamically produced during run-time. Flow dependencies can be specialized to also handle preexisting resources, such as constant values, hardware devices, and files created outside the scope of the system. Such dependency patterns follow the general form of Figure 4-25, with the exception that producer ports are now connected to *resource entities* (Section 3.3.4), describing the preexisting resources, rather than activities.

Existing resource dependencies have the following main differences from generic flows:

- Since there are no resource producers in this case, it is more natural to use the term *shared resource dependency*, rather than flow dependency to describe these relationships.

- There is no prerequisite dependency. Since resources preexist at the beginning of system execution, prerequisite requirements are automatically managed.

- There is usually no run-time identifier flow relationship among resource users. Preexisting resources are either constant data resources, or have constant identifiers. In both cases, data or identifiers can be communicated to all users at design-time, using constant definitions.

As a consequence, the typical preexisting resource dependency only needs to manage usability and sharing among users. The general discussion and design alternatives presented in Sections 4.5.1 and 4.5.4 respectively, apply here as well.

Example 6-5: Sharing a printer

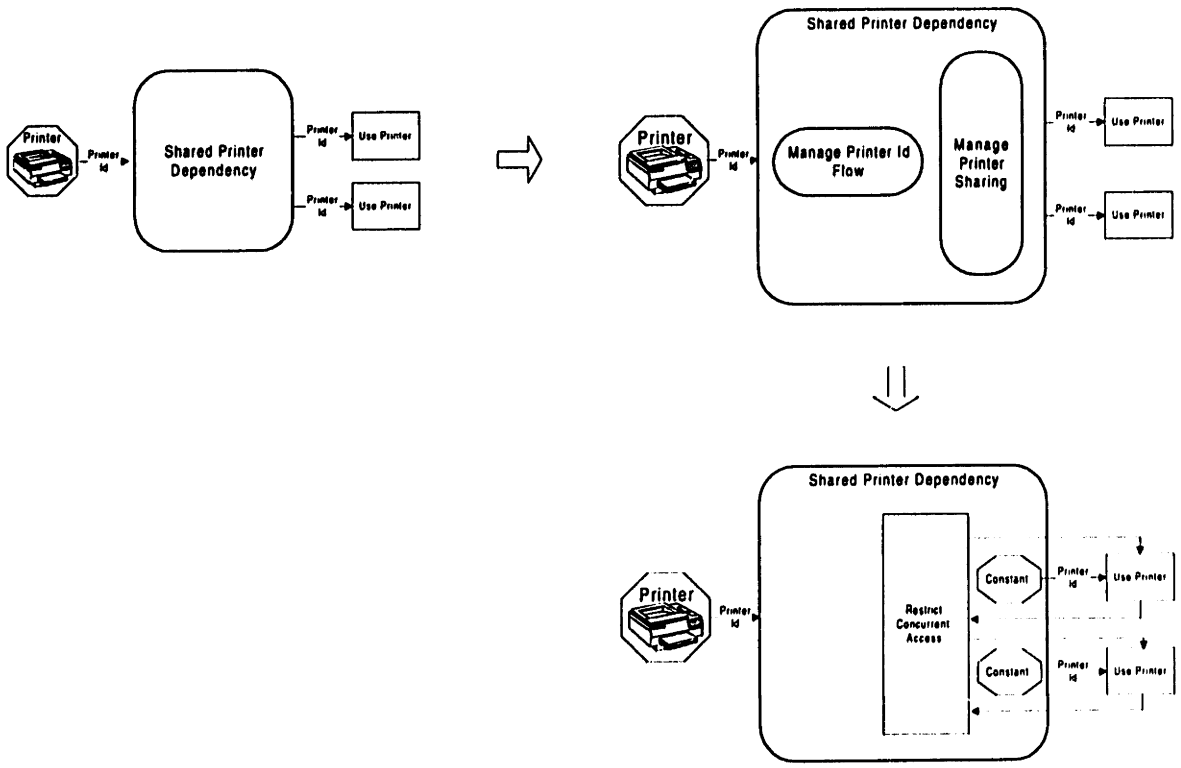


Figure 4-27: Sharing a printer can be modeled and coordinated as a special case of flow dependency.

In this example, we are interested in modeling and coordinating the sharing of a printer resource among a set of user activities. Printers are accessed by users through a device name, which is constant and known at design-time. Therefore, no run-time coordination is required for communicating the resource name. Furthermore we assume that, in this particular application, there are no usability considerations, Therefore, the only component of the shared resource dependency that requires run-time coordination is the sharing component. Printers cannot interleave the printing of multiple files. Assuming there is no built-in support for spooling, printers must be treated as nonconcurrent resources, and appropriate run-time coordination that restricts concurrent printer access must be introduced into the system.

4.7 Timing Dependencies

Timing dependencies specify constraints on the relative timing of two or more activities. The most widely used members of this dependency family are *prerequisite dependencies* (A must complete before B starts) and *mutual exclusion dependencies* (A and B cannot overlap).

Timing dependencies are used in software systems for two purposes:

- to specify implicit resource relationships
- to specify cooperation relationships among activities that share some resources

a. Specify implicit resource relationships

Implicit resource relationships arise in situations where parts of a resource flow coordination protocol have been hard-coded inside a set of components. Other parts of the protocol might be missing, and explicit coordination might be needed to manage the missing parts only. One example is a set of components for accessing a database. Each of the components contains all the functionality needed in order to access the database built into its code. The name of the database is also embedded in the components and does not appear in their interface. However, none of the components contains any support for sharing the database with other activities. In applications that require concurrent access of the database by all components, designers need to specify and manage an external mutual exclusion dependency among the components.

b. Specify cooperation relationships

Flow dependencies assume that different users of a resource are independent from one another. In many applications, however, users of a resource are cooperating in application-specific ways. Section 4.2 describes an example of such patterns of cooperation. In those cases, designers must specify additional dependencies that describe the cooperation among the users. Some of those dependencies could be other resource dependencies. Other could be timing dependencies.

In order to derive a useful family of timing dependencies we have used the following approach, based on Allen's taxonomy of time interval relationships [Allen84].

Relation	*Symmetric Relation*	*Pictorial Example*
X before Y		XXX YYY
X equal Y		XXX YYY
X meets Y		XXXYYY
X overlaps Y		XXX YYY
X during Y	X equal Y	YYYYYY XXX
X starts Y	X, Y simstart	XXX YYYYY
X finishes Y	X,Y simend	XXX YYYY

Table 4-1: Allen's taxonomy of relationships between time intervals.

In his seminal paper, Allen has enumerated all possible relationships between two time intervals (Table 4-1). An occurrence of a software activity can be represented by a time interval: *[Begin_time, End_time]*. Timing dependencies express constraints among activity occurrences. These constraints can be expressed by equivalent constraints between time intervals. Constraints can either *require* or *forbid* that a given time interval relationship holds. By enumerating "required" and "forbidden" constraints for each of Allen's time interval relationships, we get a list of potentially interesting elementary timing dependencies (Table 4-2). These dependencies can be combined to define additional, composite timing relationships. Finally, the resulting set of dependencies can be organized in a specialization hierarchy, as shown in Figure 4-28.

Allen's Relation	***"Relation Required" Dependency***	***"Relation Forbidden" Dependency***	*Comments*
X before Y	**X prerequisite Y**	**X prevents Y**	
X equal Y			Can be expressed as a composite pattern: *X,Y simstart AND* *X,Y simend*
X meets Y	**X meets Y**		Special case of prerequisite
X overlaps Y	**X overlaps Y**	**X,Y mutex**	
X during Y	**X during Y**	**X,Y mutex**	During can be expressed as a composite pattern: *X overlaps Y AND* *Y finishes X*
X starts Y	**X starts Y**		
X,Y simstart	**X,Y simstart**		
X finishes Y	**X finishes Y**		
X,Y simend	**X,Y simend**		

Table 4-2: Deriving timing dependency types from Allen's time interval relationships.

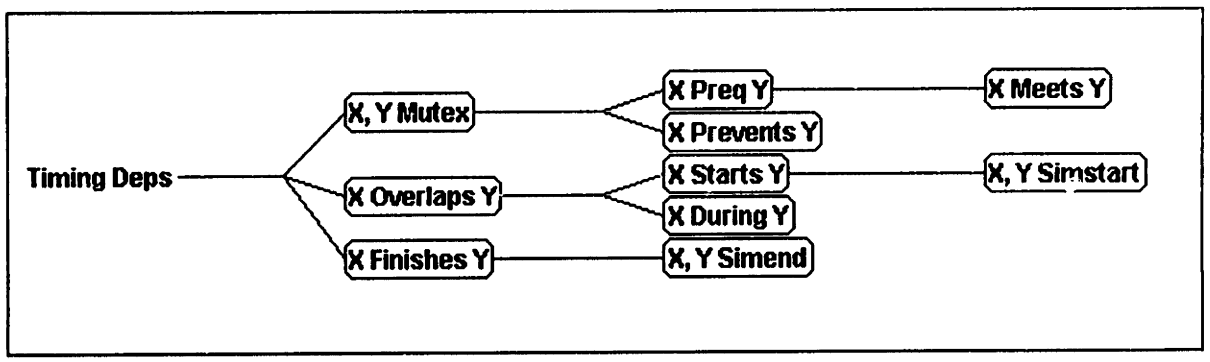


Figure 4-28: Specialization relationships between timing dependencies.

The following paragraphs describe each of the dependencies shown in Figure 4-28. For each dependency type, we describe:

- the timing constraint it specifies
- the dependency design dimensions
- the principal ways to manage it
- some situations where it might be useful

4.7.1 Mutual Exclusion Dependencies (X, Y Mutex)

Description:	Mutual exclusion dependencies among a set of activities limit the total number of activities of the set that can be executing at any one time.
Design Dimensions:	Degree of concurrency (maximum number of concurrently executing activities).
Coordination Processes:	See [Raynal86]
Typical Use:	Mutual exclusion dependencies typically arise among *competing* users who share resources with limited concurrency.

4.7.2 Prerequisite Dependencies (X Prereq Y)

Description:	Prerequisite dependencies specify that an activity X must complete execution before another activity Y begins execution.
Design Dimensions:	See Section 4.5.3
Coordination Processes:	See Section 4.5.3
Typical Use:	Prerequisites arise in two general situations: i. Between producers and consumers of some resource. A resource must be produced before it can be consumed. ii. As a special way of managing mutual exclusion dependencies. Mutual exclusion relationships can be managed by ensuring that the activities involved occur in a statically defined sequential order. The ordering can be specified by defining appropriate prerequisite relationships.

4.7.3 Prevention Dependencies (X Prevents Y)

Description:	Prevention dependencies specify that the occurrence of an activity X prevents further occurrences of another activity Y.
Design Dimensions:	- In *permanent* prevention dependencies, an occurrence of X prevents all further occurrences of Y. - In *temporary* prevention dependencies, occurrence of a third activity Z re-enables occurrences of Y.
Coordination Processes:	Prevention relationships are closely related to perishable prerequisites (see Section 4.5.3.1). As shown in Figure 4-29, every prevention dependency can be mapped to an equivalent perishable prerequisite.
Typical Use:	Prevention relationships often arise among competing activities that share some resource, where one of the competing activities (X) has higher priority, and thus the power to restrict access to (prevent) other competing activities (Y).

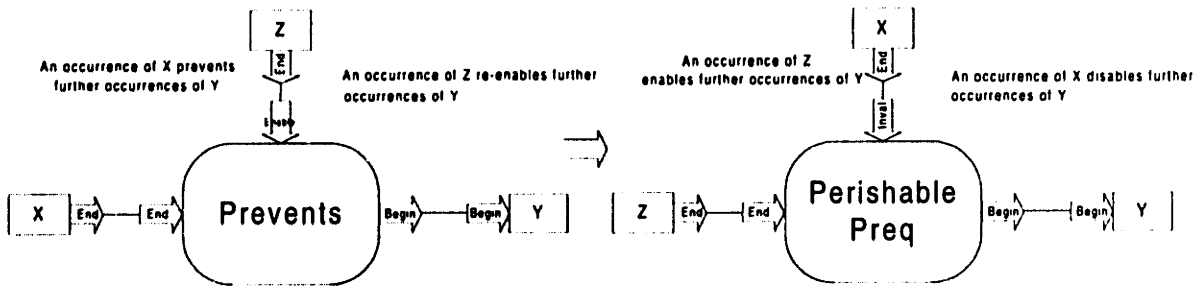


Figure 4-29 : Relationship between prevention and perishable prerequisite dependencies.

4.7.4 Meets Dependencies (X Meets Y)

Description:	Meets dependencies specify that an activity Y should begin execution after completion of another activity X
Design Dimensions:	Minimum or maximum delay between the completion of X and the initiation of Y.
Coordination Processes:	Most of the coordination processes for managing lockstep prerequisites can be used to manage this dependency. Delay parameters between X and Y can determine which alternatives are appropriate for each special case (For example, if Y must start immediately after X completes, direct transfer of control is usually preferable to loose event synchronization).

Typical Use:	Meets dependencies are a special case of prerequisite and can also be used to describe relationships between producers and users of resources. The explicit specification of maximum delay between the two activities is useful in situations where resources produced have finite lifetimes and must be used within a specified time interval.

4.7.5 Overlap Dependencies (X Overlaps Y)

Description:	Overlap dependencies specify that an activity Y can only begin execution if another activity X is already executing.
Design Dimensions:	None
Coordination Processes:	This dependency can be managed in two different ways: i. Proactively scheduling Y when X starts execution. This is equivalent to decomposing X overlaps Y to Y starts X with specified delay. ii. Waiting for X to begin execution before allowing Y to start. This is equivalent to defining a perishable prerequisite (enabled by initiation of X, invalidated by completion of X) between Y and X.
Typical Use:	Overlap relationships typically imply resource relationships between Y and X. In most cases, during its execution Y produces some resource or state required by X. Overlap dependencies occur most frequently as components of During dependencies.

4.7.6 During Dependencies (X During Y)

Description:	During dependencies specify that an activity X can only execute during the execution of another activity Y.
Design Dimensions:	None
Coordination Processes:	This dependency is a composite pattern of the following two dependencies: X Overlaps Y X can begin execution only if Y is already executing Y Finishes X Termination of Y also terminates X It can be managed by composing processes for managing its two component dependencies.
Typical Use:	During dependencies imply that X uses some resource or state generated during Y's execution. For example, a network client can only execute successfully during execution of the system's network driver.

4.7.7 Starts Dependency (X Starts Y)

Description:	Starts dependencies specify that an activity Y must start execution whenever X starts execution.
Design Dimensions:	Minimum or maximum delay between initiation of the two activities.
Coordination Processes:	Combinations of direct control flow and scheduling can be used to manage this dependency.
Typical Use:	This dependency is often used to describe application-specific patterns of cooperative resource usage or implicit resource dependencies. For example, when starting a word processor program, the printer driver is often initialized as well, in anticipation to the word processor's need for its services.

4.7.8 Simultaneity Dependency (X,Y simstart)

Description:	Simultaneity dependencies specify that all activities in a set must start execution at the same time.
Design Dimensions:	Minimum and maximum tolerances between the actual time each activity in the specified set begins execution.
Coordination Processes:	Simultaneity dependencies can be transformed into many-to-many prerequisite dependencies and managed as such (see Figure 4-30).
Typical Use:	Simultaneity dependencies are most often used to describe patterns of cooperative resource or mutual resource dependencies.

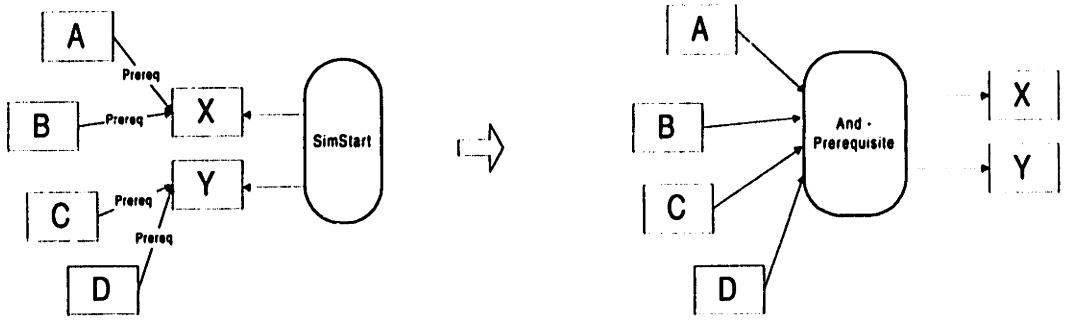


Figure 4-30: A simultaneity dependency can be transformed and managed as a composite prerequisite: Before activities X and Y can begin execution, all four prerequisite activities must first occur. Then, both X and Y can occur together.

4.7.9 Finishes Dependency (X Finishes Y)

Description:	Finishes dependencies specify that completion of an activity X also causes activity Y to terminate execution.
Design Dimensions:	Minimum or maximum delay between completion of X and termination of Y.
Coordination Processes:	Termination of the process that executes Y using machine-specific system primitives.
Typical Use:	This dependency is most often used to specify application termination relationships (Figure 4-31).

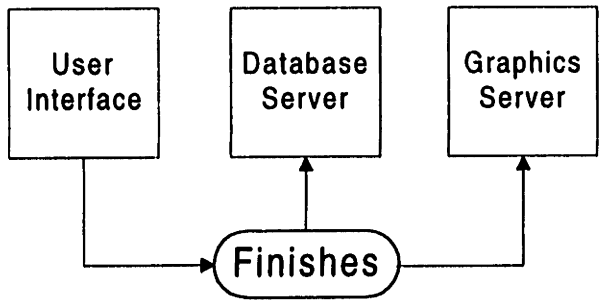


Figure 4-31: Termination of the user-interface also requires termination of the database and graphics servers.

4.7.10 Simultaneous End Dependency (X, Y Simend)

Description:	Simultaneous end dependencies specify that all activities in a set must terminate if any of them completes execution.
Design Dimensions:	Minimum or maximum tolerances between the actual time each member of the specified set terminates.
Coordination Processes:	*Centralized:* Each activity in the set sends a signal to a monitor process upon termination. The monitor process terminates all other activities in the set. *Decentralized:* Terminating activities generate an event. All participant activities periodically check for that event and terminate themselves if they detect it.
Typical Use:	*Speculative concurrency*: Multiple worker activities are jointly or independently working on a problem. All of them terminate if at least one of them arrives at a solution.

4.8 Composite Dependencies

The approach we have used in most of this chapter is based on decomposing complex relationships into sets of *independent* simpler relationships, enumerating ways of managing each of the simpler relationships, and combining the pieces to construct coordination processes for the more complex relationships. This has resulted in a taxonomy of elementary dependency types and their respective coordination processes.

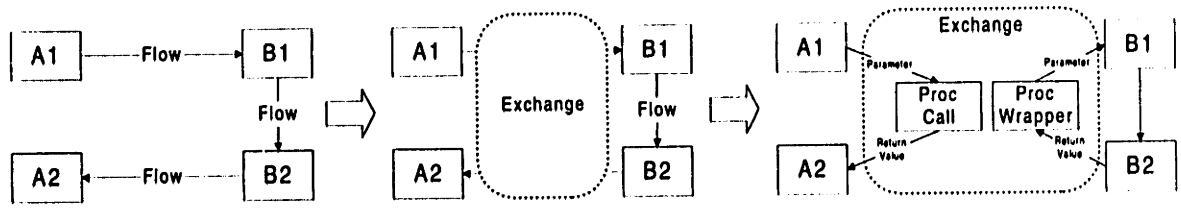
Although the definition of a vocabulary of simple dependencies, that can be used and managed independently of one another is a very desirable objective (see Section 4.1), throughout the chapter we have observed that in several cases there exist *joint* coordination processes for managing together more than one dependencies. For example,

replication is a process that jointly manages accessibility and sharing issues in one-to-many flow dependencies, often more efficiently than if the two issues had been managed separately.

It would be very useful to begin to classify such composite dependency patterns, and the specialized joint coordination processes for managing them. When encountering such patterns, designers will have a choice between decomposing them into their components and managing each component independently, or using a joint managing process for the entire pattern,

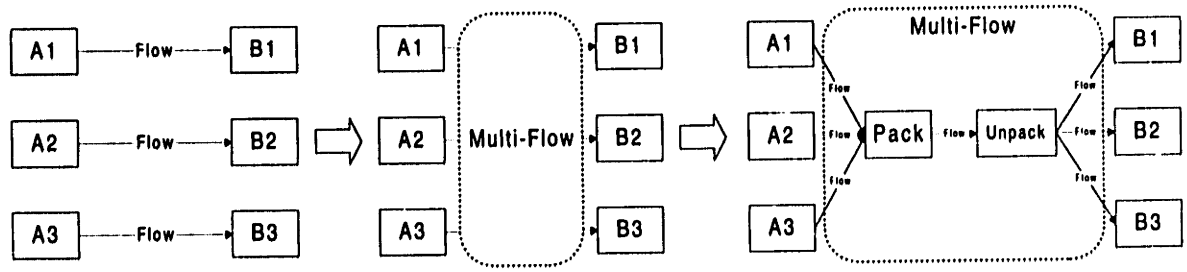
SYNOPSIS provides full support for defining and storing composite dependencies. Although a systematic classification of composite dependencies lies beyond the scope of this thesis, the rest of the section provides evidence of the usefulness of this task by describing a few commonly occurring complex patterns of flow dependencies, for which special joint coordination processes have been developed.

4.8.1 Exchange Dependencies



Pairs of flow dependencies in opposite directions can often be efficiently managed by combining them into a single procedure call. One of the flows can be managed by the flow of parameters from caller to callee, while the opposite flow can be managed by the flow of the return value from callee to caller. We call such composite pairs of flow dependencies, exchange dependencies.

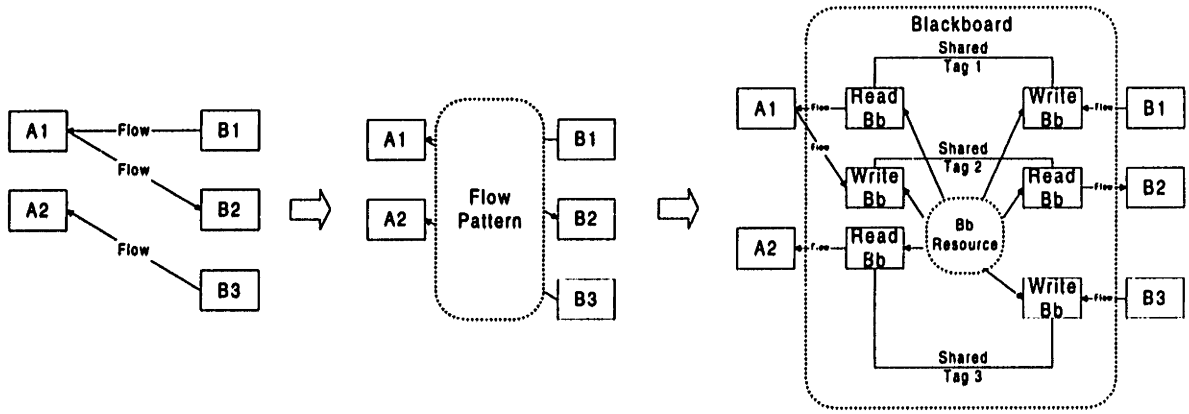
4.8.2 Multiple Unidirectional Flows



Sets of flow dependencies in the same direction can often be managed very efficiently by collecting them together, packing them into composite structures, and sending them using a single flow. At the other end, they are unpacked and distributed to their users. Such a

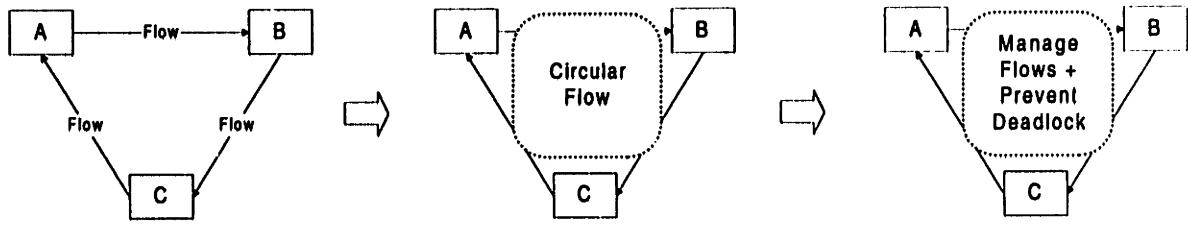
technique is used when sets of small messages are collected together into larger packets and sent through the network in one shot.

4.8.3 Arbitrary Static Flow Patterns



Arbitrary sets of flows can be managed together using a divisible, associative access repository of information, often called a *blackboard*. Data items are written and read from the blackboard using associative tags. A single blackboard resource is able to concurrently manage multiple flow transactions provided that each transaction uses its own unique tag. The language Linda [Carriero89, Gelernter92] is built around the blackboard model of flow management. Linda calls its blackboard object *tuple space*.

4.8.4 Circular Flows



Sets of flow dependencies that form a cycle, create the possibility for deadlock conditions among the interdependent activities. For that reason, we can define special composite coordination processes that jointly manage sets of circular flows. Such coordination processes, in addition to managing each flow will install additional coordination that will prevent deadlock.

4.9 The Way Ahead

This chapter has introduced a vocabulary of dependency types for describing software component interconnection relationships and an associated design space of coordination processes for managing each dependency type. Combined with the linguistic support of SYNOPSIS, we are now well-armed in order to:

- describe the functional pieces of new applications using activities
- describe the interconnection requirements of new applications using dependencies
- associate activities to code-level software components
- manage dependencies by introducing coordination processes

The piece of the puzzle that is still missing is how activities and coordination processes can be integrated into executable applications. Chapter 5 is devoted to a discussion of the issues involved, and presents an algorithm that can automate the integration process.

Chapter 6 describes SYNTHESIS, a prototype implementation of a software development tool based on the ideas described in this thesis. It also discusses a number of experiments performed using SYNTHESIS, in order to test the feasibility and usefulness of our approach.

Chapter 5

Transforming Software Architectures into Executable Applications

At the beginning of this thesis we made the case for separating the description of an application's functional components from that of interconnection relationships among the components. We argued that such a separation will facilitate the development of new applications from existing software components, and proposed an outline of a process for developing such applications. The process is based on applying a series of transformations to an architectural description of a target application. Chapter 3 introduced SYNOPSIS, an architectural description language for describing software applications. This chapter focuses on the transformations that can be applied to SYNOPSIS architectural diagrams, in order to generate executable applications. It presents algorithms for assisting, and in some cases automating, each transformation. It integrates those algorithms into a concrete implementation of the process proposed in Chapter 2. Finally, it concludes with a detailed walkthrough of the process for a concrete example.

5.1 Introduction

In Chapter 2 we proposed an outline of a process for developing new software applications from existing components. The aim of this process is to reduce the cost of composing software components by minimizing the need for ad-hoc, user-written coordination code to bridge component mismatches (Figure 5-1).

The process is based on constructing software architecture descriptions, which clearly distinguish and separate the main functional pieces of an application from their interconnection relationships in the context of the application. The process then allows

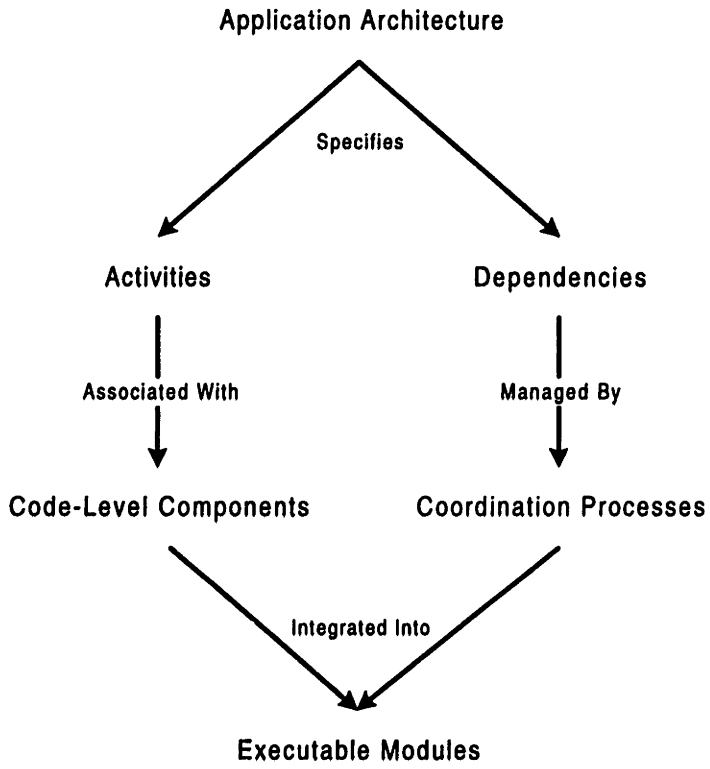


Figure 5-1: A high-level view of a process for developing component-based software applications (Repeated from Figure 2-9).

designers to generate executable applications by applying a set of transformations to these architectural descriptions.

Chapter 3 focused on the entities of the process (the "nodes" of Figure 5-1). It introduced SYNOPSIS, an architectural description language that provides the linguistic tools necessary in order to build such descriptions. SYNOPSIS describes software applications as graphs of activities interconnected through dependencies (for example, see Figure 3-1). Activities specify the main functional pieces of an application. Dependencies specify interconnection relationships among activities.

This chapter focuses on the transformation steps of the process (the "arrows" of Figure 5-1). The practical value of the proposed process will depend on how easy it is, both to construct the initial architectural diagrams, and to perform the operations that transform them to executable code. At best, we would like as much as possible of the process to be assisted, or automated, by computerized tools.

For each of the "arrows" in Figure 5-1, section 5.2 will describe:

- the transformation involved

- opportunities for assisting the transformation by computer
- system support necessary for assisting the transformation

Section 5.3 will integrate the described system support functions into an algorithm for semi-automatically generating executable applications from SYNOPSIS descriptions. Finally, section 5.4 will illustrate its use on a simple example.

5.2 Constructing and Transforming Architectural Diagrams

There are four stages in the process of Figure 5-1:

- Constructing application architecture diagrams
- Iteratively specializing generic activities
- Iteratively managing unmanaged dependencies
- Integrating executable design elements into code modules

The following sections will discuss each stage in detail. For each stage, we will explore opportunities for assisting or automating it, and will describe system support necessary for implementing these opportunities.

5.2.1 Constructing Application Architecture Diagrams

The first step in the process involves the specification of an application architecture as a set of activities interconnected through dependencies. It is important to be able to generate application architectures rapidly and correctly. The basic requirements for performing this step include:

- Linguistic support for defining and checking architectural diagrams
- Design support for leveraging the construction of correct diagrams

a. Linguistic support

SYNOPSIS provides a set of graphical abstractions for defining activities and dependencies, as well as the linguistic means to interconnect them (ports and connectors, see Section 3.3). Furthermore, it supports a compatibility checking mechanism that is able to detect a number of inconsistencies when connecting elements together (Section 3.4).

b. Design support

Apart from linguistic support for specifying activities and dependencies, the construction of application architecture diagrams can be facilitated by:

- assistance on how to decompose an application to simpler functional pieces
- assistance on how to represent interdependencies among application functional pieces

Application decomposition can be facilitated by the reuse of generic decompositions for frequently occurring problems. This requires the creation of *process repositories*, or *process handbooks*, for storing, searching, and reusing architectural design fragments. A related project, which is focusing on developing a handbook of organizational processes is described in [Malone93, Dellarocas94].

SYNOPSIS provides a number of mechanisms for supporting the construction of process repositories. Architectural patterns can be expressed as composite activities. Through the mechanism of *entity specialization* (Section 3.5), sets of related composite activities can be organized and stored in a specialization hierarchy. Specialization hierarchies are similar to class hierarchies in object-oriented system, and can be used as the basis for structuring repositories of reusable architectural patterns.

The problem of specifying interdependency patterns among application activities can be similarly facilitated by a repository of dependency types for frequently occurring patterns of interaction. One of the hypotheses of this thesis is that interconnection relationships can be described using a relatively narrow set of concepts, orthogonal to the problem domain of most applications. For that reason, we have attempted to define a standardized, but extensible, *vocabulary of dependency types*, that can be used by designers to express the interaction requirements in their applications. The specialization mechanism of SYNOPSIS can be used to store and structure the vocabulary of dependencies. The vocabulary of dependencies is described in Chapter 4.

5.2.2 Specializing Generic Activities

SYNOPSIS activities can be *generic* or *executable* (see Section 3.3.1). This enables designers to specify the architecture of their applications at any desired level of abstraction. However, in order for executable code to be generated, all generic activities must first be replaced by executable specializations. Executable atomic activities, also called *primitive activities*, are associated with some code-level software component, such as a source code module, or an executable program. Composite executable activities decompose into sets of executable elements. Therefore, the step of replacing generic activities results in the eventual association of all activities in SYNOPSIS architectural diagrams with sets of code-level software components.

Although this step is very important, the methodology proposed in this thesis does not provide specific design guidance on how to perform it. The responsibility for *locating* and *selecting* the most appropriate code-level components is left with the designer. We believe that, of all problems related to software reuse, location of appropriate components is the one that is currently closer to a satisfactory solution. Several researchers are

developing technologies for collecting software components into *component libraries*, easily accessible to the community of designers [IMSL87, Dongarra87]. Furthermore, new technologies, such as the Internet, are making a growing number of software component repositories readily accessible to designers.

Nevertheless, the spirit of the methodology does provide some leverage for the component selection process because each activity can be associated to a software component independently of any other activity. Also, designers do not have to worry about the specific form of components and their interfaces. Handling of potential mismatches between component interfaces is completely contained in the coordination processes that manage dependencies among components. This facilitates the selection process because designers need only care about whether a given component contains the functionality required by its associated activity and not about how it will fit together with other components already selected.

5.2.3 Specializing Generic Dependencies

One of the novel contributions of this work is the argument that the functional pieces of a software application and their interdependencies should be both specified and implemented independently of one another. In order to generate an executable system, generic dependencies must be replaced by executable specializations. Executable dependencies are either directly associated with a software connector, implementing a low-level interconnection mechanism, or with a coordination process, defined as a pattern of simpler dependencies and activities (Section 3.3.2). A large part of this work concentrates on providing support for assisting, and in some cases automating, the replacement of dependencies with appropriate specializations.

As with the previous steps, there are two basic requirements for performing this step:

- *Linguistic support:* Provide adequate abstractions for defining dependencies.
- *Design support:* Assist designers in selecting a software connector or coordination process that manages a given dependency.

a. Linguistic support

SYNOPSIS represents coordination processes and software connectors as attributes of dependencies (see Section 3.3.2). Coordination processes provide a single home for specifying all the pieces of an interaction protocol. In contrast, traditional programming languages usually force the description of interaction protocols to be distributed among the interacting components.

Coordination processes are defined as compositions of lower-level dependencies and activities. Furthermore, coordination processes can be *generic* or *executable*. Generic

coordination processes have at least one generic activity, or unmanaged dependency, in their decomposition. Coordination processes are executable if all their elements are executable.

Software connectors correspond to low-level interconnection mechanism, directly supported by programming languages and operating systems (e.g. procedure calls). Dependencies associated with a software connector are executable.

b. Design support

i. Assisting the selection of coordination processes

Simply being able to define a coordination process does not tell designers what should be contained in that process. The biggest difficulty in building applications from existing components lies exactly in designing and implementing such *coordination software* that bridges mismatches and manages interconnection requirements. Using current technologies, designers had to almost always build that software from scratch.

One of the contributions of this work is the development of a design space of coordination processes for each element of the vocabulary of dependencies. Chapter 4 contains a detailed description of the design space. The design space maps each type of dependency to a family of alternative coordination processes for managing it. A particular coordination process is selected from that family by specifying the values of a relatively small number of additional parameters called *design dimensions*. For example, a coordination process for managing a data flow dependency can be selected by specifying the type of carrier resource (e.g. shared memory, file, pipe) and the paradigm (push, pull, peer, hierarchy) for managing the embedded prerequisite (see Section 4.6.2). Some of the alternatives can be automatically ruled out by compatibility constraints. For example, when designing a data flow between two components running under UNIX, only data transport mechanisms supported by UNIX can be considered as alternatives. This reduces the selection of a coordination process to a routine selection of a relatively small number of design parameters.

Most of the coordination processes introduced in Chapter 4 are defined at the generic level. This means that they are defined as sets of lower-level generic activities and unmanaged dependencies. For example, the generic process for managing a one-to-one data flow dependency (Figure 4-21) consists of two lower-level generic activities (Manage Usability, Transport Data) and a number of unmanaged flow dependencies. In order to integrate that process in an executable application, each of those generic decomposition elements has to be specialized in turn. Therefore, the previous candidate selection process has to be *recursively* applied to all decomposition elements as well. The design process completes when all activities and dependencies introduced are executable. Executable activities have associations with code-level software components. Executable dependencies are associated to code-level software connectors, directly supported by

specific programming language or operating system mechanisms (e.g. C procedure calls, or UNIX pipe protocols).

It is obvious that the above design process can be greatly assisted, or even automated, by computer. Assuming that there is an on-line repository of increasingly specialized dependency types, a generic dependency appearing in a SYNOPSIS diagram can be easily managed as follows: A design assistant automatically searches the repository of managed specializations of the generic dependency. It rules out dependencies whose associated coordination processes are rejected by the compatibility checking algorithm of Figure 3-15. It then asks designers to simply select one of the dependencies that pass the compatibility test. Alternatively, it can present designers with a set of design dimensions to be specified. If the coordination process associated to the selected dependency decomposes into lower-level unmanaged dependencies or generic activities, the selection process is recursively applied to each of them. Instead of using recursion, a *to-do list* can be used to iteratively store encountered generic activities and unmanaged dependencies, and then retrieve and handle them until the list becomes empty. Figure 5-2 shows a flowchart of such an algorithm.

The process described by Figure 5-2 can be fully automated by allowing designers to specify *evaluation functions*, or constraints, that enable the computer to automatically rank candidate alternatives for each transformation. Such evaluation functions could be constraints on specific process attributes (e.g. *"all coordination processes must adhere to client/server organizations"*) or functions related to the computational cost of coordination processes. Evaluation functions can further restrict the number of candidate coordination processes presented to designers. In some cases, they might restrict the candidates to one, in which case the selection process becomes fully automatic.

Since the process relies on the existence of a repository of coordination processes, there might be situations for which no compatible coordination process has been stored in the repository. For example, when managing a usability dependency which requires conversion of strings to integers in Visual Basic, there might be no such specialization of the conversion activity in the repository. In such cases, the system asks the users to define a new activity specialization with that functionality. The specialization typically does not require more than a few lines of code, and becomes a permanent part of the repository. In that manner, repositories of design elements can be incrementally extended and eventually become rich enough to be able to handle a large number of practical cases.

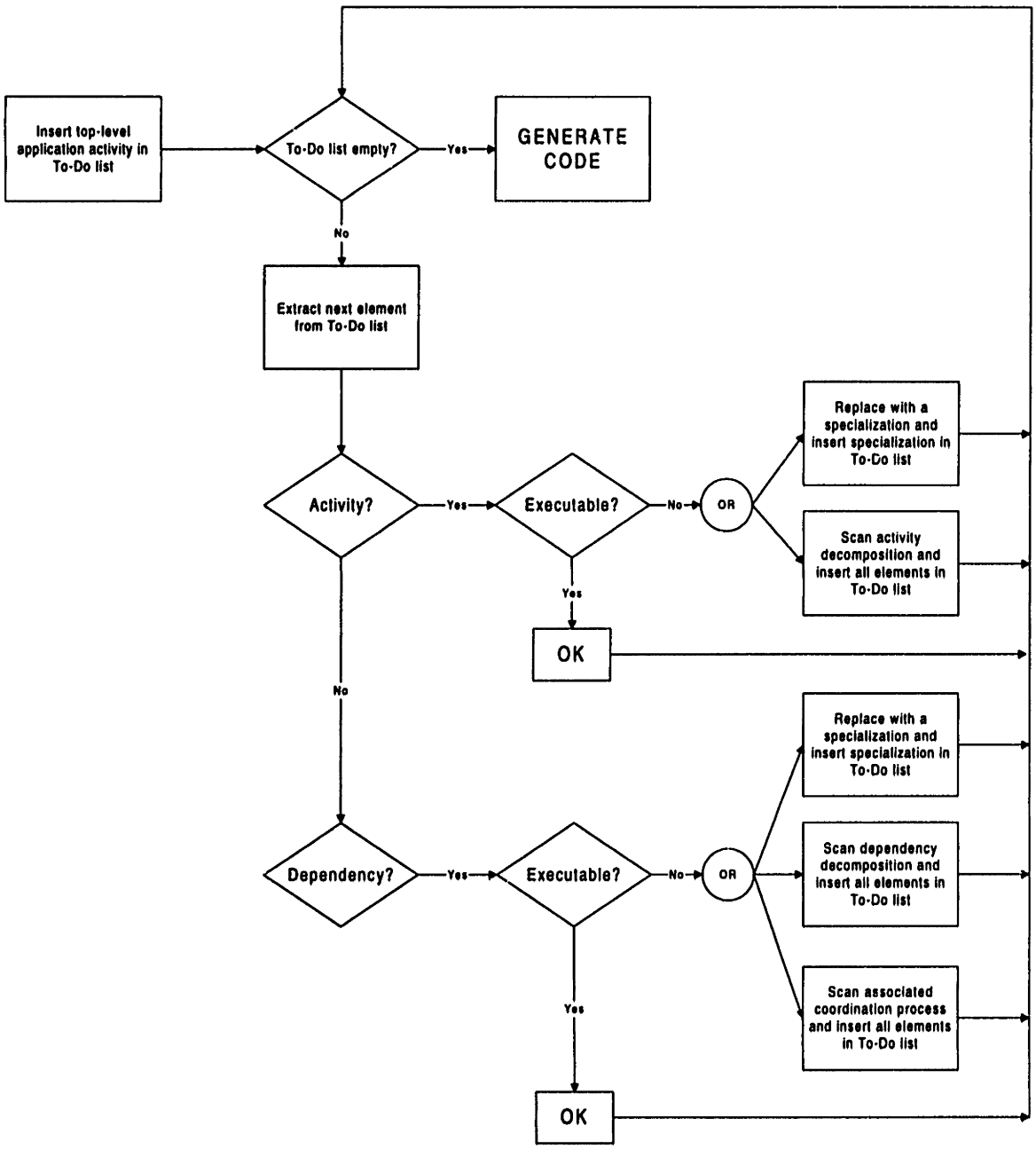


Figure 5-2: An algorithm for iteratively specializing generic elements in SYNOPSIS architectural descriptions.

ii. Decoupling interface dependencies

The algorithm of Figure 5-2 relies on the assumption that every dependency in a SYNOPSIS diagram can be managed independently of every other dependency (unless the designer has explicitly combined a number of dependencies into a composite dependency pattern). This assumption is equivalent to the assumption that the connections of an activity port to other parts of the application can be managed independently of the connections of every other port of the same activity.

This assumption, if it holds, has a number of desirable consequences:

- it enables designers to describe application architectures independently of the particular interfaces of the underlying components
- it enables a useful repository of coordination processes to be created by storing coordination processes for managing elementary activity patterns.

Unfortunately, software components built with current technologies contain *interface dependencies* among their interface elements which, unless properly decoupled, would force connections of sets of activity ports to be jointly managed.

Example 5-1:

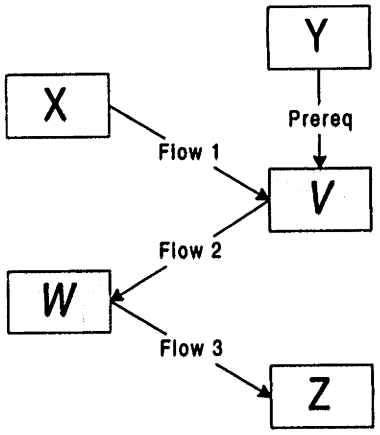


Figure 5-3: An example application where interface dependencies force joint management of flow dependencies.

Figure 5-3 shows a fragment of an application architecture. Activities X, Y, and Z are associated with source code procedures. Activities V and W are associated with remote servers that can be called from source code using remote procedure calls (RPCs). At best, we would like to be able to manage each of the four dependencies shown in the diagram independently of one another. This would allow a simple repository of one-to-one

coordination processes to handle this example. However, the semantics of RPC interfaces place obstacles to this goal.

RPC interfaces expect to receive all their input arguments (plus control) at the same time, packaged inside a single RPC call. Moreover, they return all output values to the point of call through that same interface as well.

Managing dependencies independently of one another requires that their coordination processes do not share any code. Therefore, dependency Flow 1 should be able to communicate its value to the V server without relying on any code introduced by the management of any other dependency. Likewise, dependency Flow 2 should be able to access the result of invoking the server independently of how Flow 1 has been managed. Finally, dependency Prereq should be able to ensure that Y occurs before V, independently of any code introduced by the management of either Flow 1 or Flow 2. Unfortunately, RPC semantics force coordination processes for all three dependencies to share a common step (the RPC call). Thus, management of the three dependencies has to be done jointly.

For similar reasons, dependencies Flow 2 and Flow 3 must be jointly managed. Finally, all four dependencies in the diagram have to be looked at jointly, in order to be properly managed. In addition to requiring additional machinery for automatically detecting them, the existence of such implementation-dependent *"dependencies among dependencies"* would require explicit support for arbitrarily complex composite dependency patterns in the "design handbook" repository.

Example 5-2:

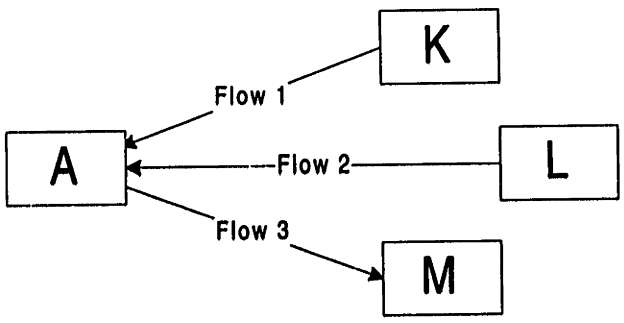


Figure 5-4: Another example application fragment with interface dependencies.

Figure 5-4 shows another fragment of an application architecture. In this example, A is associated with an executable program for the Microsoft Windows environment, which expects the existence of a DDE Server with specific name and interface. In this application, we are trying to connect the executable with three independent source

procedures, K, L, and M, each of which provides (or receives) one element of the expected DDE Server interface.

In order to properly connect the source procedures with the executable, they will have to be wrapped together into a DDE Server with a single interface, compatible to the one expected by the executable. This requires the joint management of all three flows appearing in Figure 5-4.

Examples 5-1 and 5-2 have demonstrated situations in which properties of specific interface types force the joint management of arbitrarily complex dependency patterns. Such "dependencies among dependencies" require both additional machinery in order to be automatically detected and more complex repositories of coordination processes in order to be semi-automatically managed.

Interface dependencies are related to the shortcomings of current-technology code-level components in separating interconnection assumptions from the implementation of a component's core functionality (see Chapter 2 for a detailed discussion). They are intrinsic properties of the code-level components chosen to implement atomic activities. For that reason, they should be handled at a level orthogonal to that of managing dependencies among activities.

One way of handling interface dependencies is by introducing additional activities that attempt to decouple them. In other words, *before* attempting to manage any dependency, scan primitive activities and detect interface dependencies among their ports. Whenever such dependencies are found, replace the original primitive activity with a composite *augmented activity* which includes activities for decoupling ports from one another, allowing them to be independently managed.

Detecting interface dependencies is easy, because they are inherent properties of specific interface types. CDL definitions associated with every primitive activity contain information about the component's *provided* interfaces, as well as other interfaces *expected* by the component (see Section 3.3.1.2).

Interface definitions involve a set of input and output data elements (for example, `proc foo( in a:Integer, out b:String); )`. Each of those elements is mapped to a corresponding atomic port of the primitive activity. Interface elements are dependent on one another because they typically have to be combined together into a single interface call (e.g. a procedure call), or a single interface header (e.g. a procedure header).

One very general way to decouple interface dependencies is by transforming complex interfaces to and from sets of local variables. Each local variable will store one interface element. Since local variables can be independently read or written, this set of

transformations will enable each input resource to be independently produced and each output resource to be independently consumed.

The transformations require the introduction of two sets of mediator activities *around* primitive activities with interface dependencies:

a. *Callers:* Activities that read (write) a set of independent local variables and construct a corresponding call to a given composite interface.

b. *Wrappers:* Activities that make a given composite interface available to the application, and demultiplex its elements into sets of independent local variables.

Using callers and wrappers, the process of decoupling interface dependencies among activity ports can be expressed as follows:

For each primitive activity with interface dependencies, structurally replace it with an *augmented activity* which, in addition to the original activity, contains the following mediator activities:

- A caller activity for each interface provided by the component. The caller activity connects to all ports originally associated with the provided interface of the component (Figure 5-5). Hence, it allows each dependency originally connected to the activity to independently read (write) its associated resource to (from) a local variable. The caller activity is then responsible for assembling all resources into the appropriate interface call.

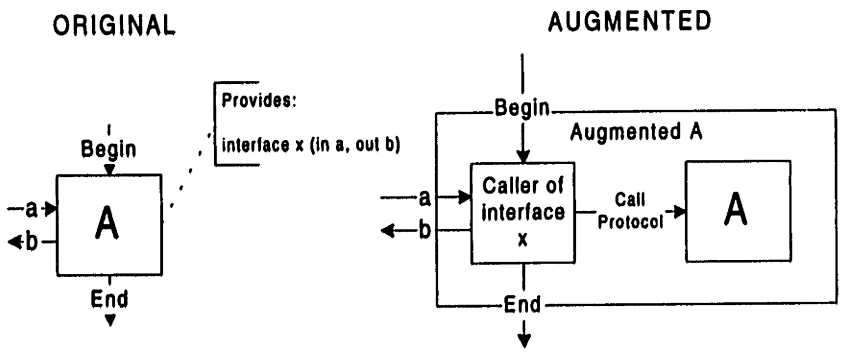


Figure 5-5: Augmenting an activity by introducing a caller.

- A wrapper activity for each interface expected by the component. Wrapper activities connect to all ports originally associated with each expected interface (Figure 5-6). Wrapper activities define headers of composite interfaces called by other components. In addition, they store each element contained in those headers to independent local variables.

Callers and Wrappers are specified per interface type. In fact, in order to support a new component kind, designers must specify appropriate caller and wrapper activities for its associated interface type.

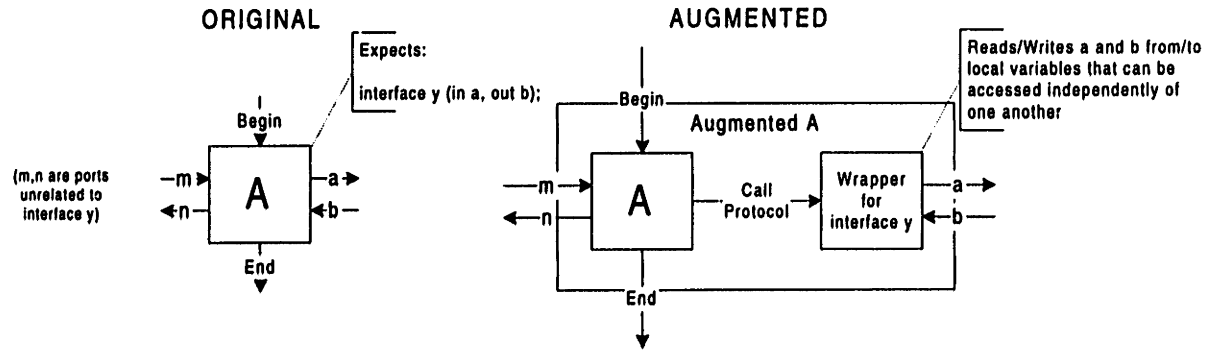


Figure 5-6: Augmenting an activity by introducing a wrapper.

Example 5-3:

This example will demonstrate how callers and wrappers can be defined for a number of common interface types.

a. Source procedure interfaces

If a component *provides* a source procedure interface, it can be invoked from inside other blocks of code by simple procedure calls. The default semantics of procedure calls enable them to receive their input parameters, and leave their output parameters to sets of independent variables. For example, in the C language call statement:

```
foo(a, b);
```

`a` and `b` are local variables that can be given values by independent statements preceding the call. As a consequence, provided procedure interfaces need not be augmented.

If a component *expects* a source procedure interface, this means that there is a call to a procedure (defined externally to the component) with the expected interface from within the code of the component. Therefore,

- a procedure definition with the expected name and parameter list must exist in the final application, and
- the code of this procedure must allow each element of its parameter list to be independently accessed by each activity connected to the corresponding atomic port of the calling component.

Such components are augmented by the introduction of *procedure wrapper* activities. Procedure wrappers translate to source code headers of procedure calls with the specified name and parameter list. Activities originally connected to the ports of an expected procedure interface, will be connected to the wrapper after augmentation takes place. For example, in Figure 5-7, the component associated with activity A contains an internal call to procedure `foo`. In the application diagram shown, the two parameters `x` and `y` passed to procedure `foo` by component A, must be independently accessed by components B and C. During code generation, calls to the components associated with components B and C will be packaged inside the body of the procedure defined by the wrapper. Therefore, they will be able to independently access each procedure call parameter through local variables.

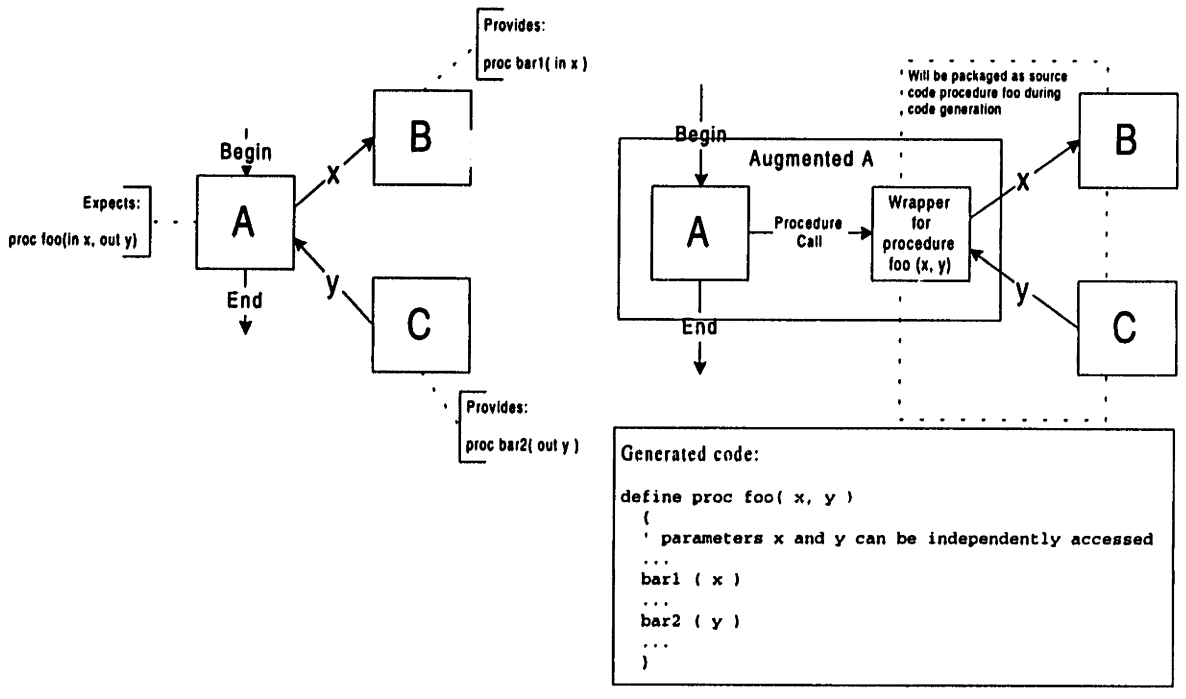


Figure 5-7: Introducing a procedure wrapper.

b. Executable program interfaces

Components that *provide* executable program interfaces, correspond to executable program files. Caller activities for executable components are source code procedures which independently receive all command line parameters necessary to invoke the executable and construct the appropriate operating system invocation call (Figure 5-8).

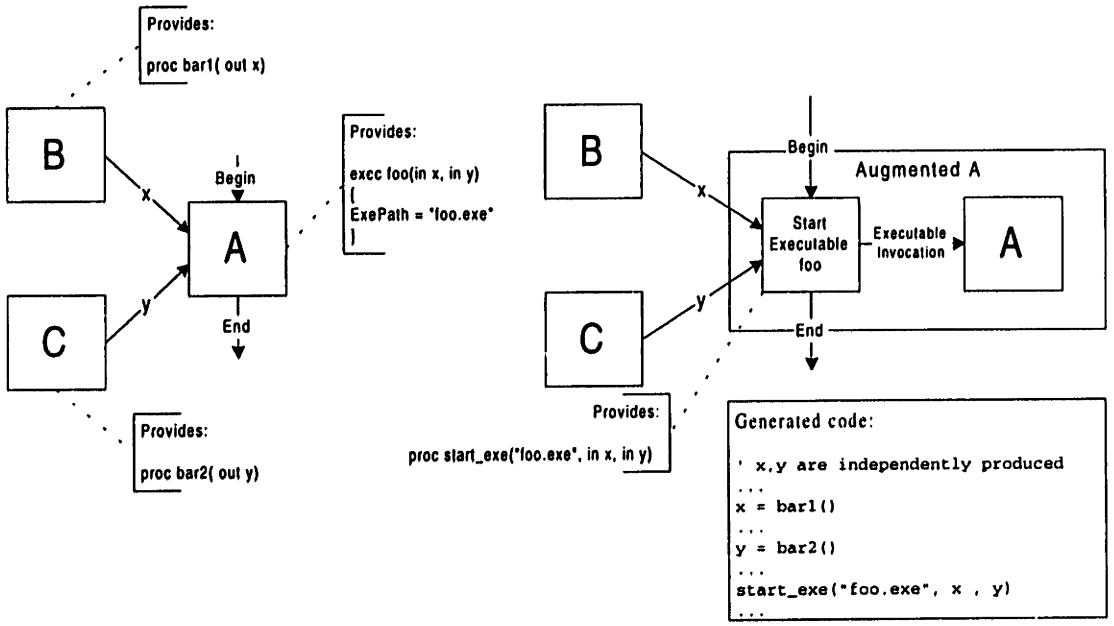


Figure 5-8: Introducing an executable caller.

If a component *expects* an executable program interface, this means that an executable program with the expected name and interface is invoked from within the code of the component. Such components are augmented by the introduction of *executable wrapper* activities. Executable wrapper activities translate to main procedure headers for executable programs with the expected name and parameter list. In addition, they introduce system-specific activities which read the command line argument structure passed to the main procedure by the operating system, and leave each passed argument to a different local variable. Activities originally connected to the ports of an expected executable interface, will be connected to the wrapper after augmentation takes place (Figure 5-9). During code generation, those activities will be packaged inside the body of the main procedure of the executable defined by the wrapper. Therefore, they will be able to indedendently access each command line argument through local variables.

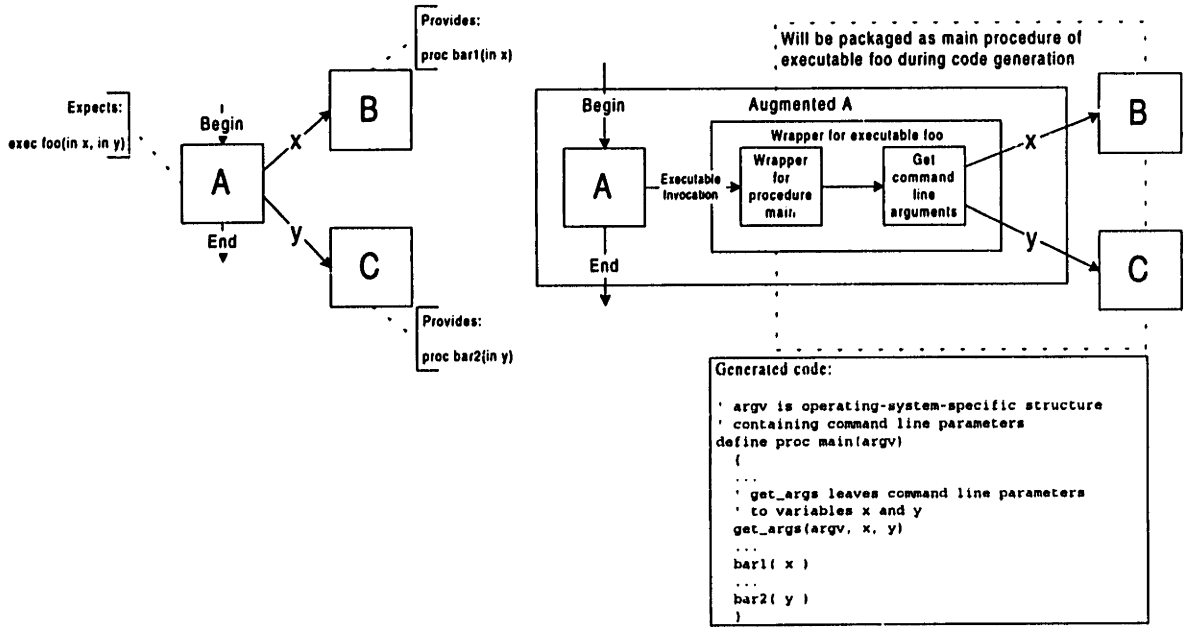


Figure 5-9: Introducing an executable wrapper.

c. Graphical user interface functions

Some programs, designed for interactive use, activate certain functions when users press a key sequence. For example, a text editor opens a file when users press CTRL-O, followed by the filename, followed by newline. In order to integrate such programs into larger applications, we view them as components that provide a special kind of interface, called a graphical-user-interface-function (*gui-function*). Gui functions specify lists of input parameters, like any other interface. They also specify a format string, to which input parameters will be embedded in order to form the activation key sequence, and a window name, to which the activation key sequence should be send in order to activate the desired function.

Caller activities for gui functions are source code procedures which receive the target window name, the format string, and all input parameters, and use operating system calls in order to send the activation key sequence to the target window (Figure 5-10).

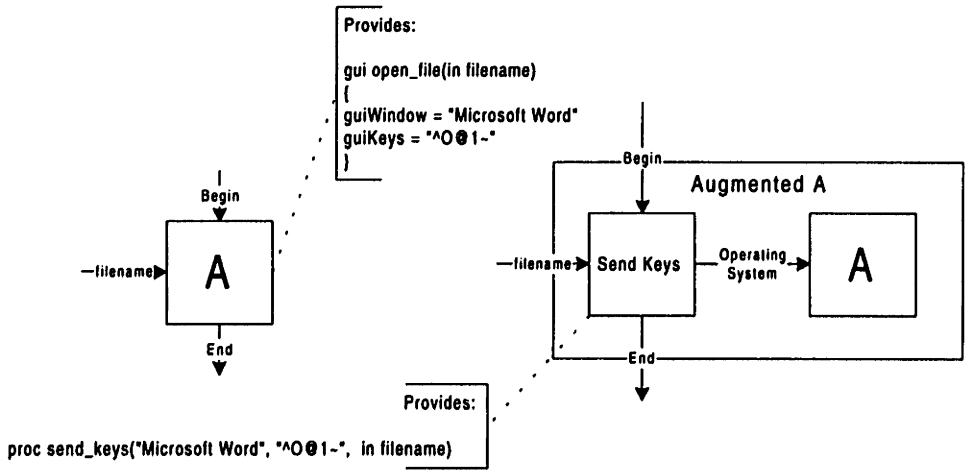


Figure 5-10: Introducing a graphical user interface function caller.

There is one special case where the explicit introduction of caller and wrapper activities might lead to inefficiencies, or even errors. That is when two of more components with perfectly matching composite interfaces are connected to each other. In such cases, the composite pattern of dependencies that specifies the connections among those interface elements is automatically managed. In order to complete the support for this step, design assistants should first check for this special case.

Example 5-4:

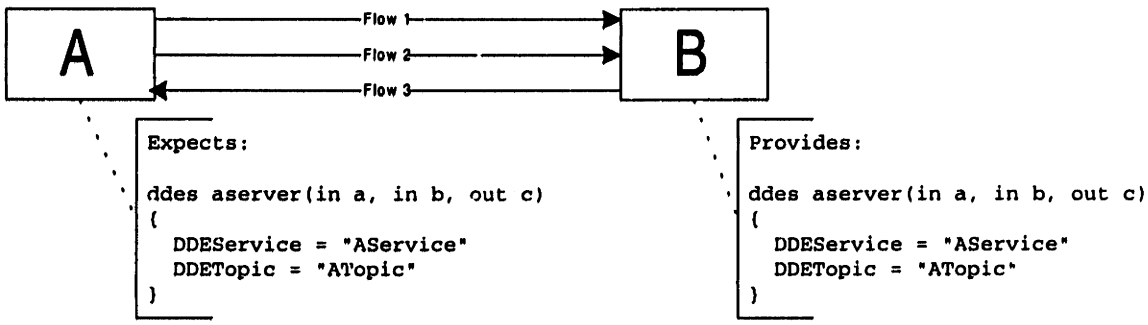


Figure 5-11: An example of two perfectly matching interfaces.

Figure 5-11 shows a fragment of an application architecture in which two activities are connected through a number of flow dependencies. One activity is an executable program expecting a DDE Server and the other is a DDE Server with the exact name and interface expected by the executable. In this special case, all three flow dependencies between the two activities are automatically managed without the need for additional coordination software. Design assistants should be able to check for such special cases and suppress the introduction of caller and wrapper activities when they detect them.

5.2.4 Integrating Executable Design Elements into Code Modules

The "raw material" of our component-based application development process is a set of code-level software components, in source or executable form. The end product is also a set of code-level components, consisting of the original components plus some additional modules containing the coordination software that manages interdependencies among the original components. The distinction between activities and dependencies leads to a very useful intermediate representation, which facilitates the design of the coordination software. Eventually, however, the original components plus the new activities and dependencies introduced by the process of design, must be integrated into sets of source or executable modules. This section describes this transformation step in more detail and shows how it can be completely automated.

Following the construction of an initial SYNOPSIS architectural diagram for a new application, designers proceed by iteratively specializing activities and dependencies. The process terminates when all elements of the diagram have been replaced by an executable specialization.

Executable activities have direct associations with code-level software components, such as source or executable code modules. Executable dependencies are associated with built-in language and operating system interconnection processes, such as the following:

- *Sequentialization processes* (Seq), managing prerequisite dependencies (see section 4.6.1). Seq coordination mechanisms specify that the activities at their endpoints will be packaged inside the same sequential code block and that the precedent activities will always be placed before the consequent activities inside that code block.

- *Local variable transfer processes*, managing data flow dependencies. Local variable coordination mechanisms have the same semantics as Seq. In addition, they specify that a data item is being transferred from the precedents to the consequents through some automatically named local variable.

- Other *built-in language or operating system interconnection mechanisms*, such as remote procedure calls etc. These processes are automatically provided by the

infrastructure of the target configuration and do not translate into tangible coordination code. Therefore, these primitive coordination processes serve mostly as comments and can be ignored in the discussion that follows.

Seq and local variable coordination processes divide the application graph into a set of partially ordered subgraphs of activities. Each subgraph contains activities to be packaged into the same sequential code module.

In order to generate an executable system from a SYNOPSIS graph, the system must be able to translate that graph into a set of modules in one or more programming languages. The translation process has two stages:

- Connect all sequential modules to control
- Generate executable code

a. Connect all sequential modules to control

In order to begin execution, every software module must receive control from somewhere. This is an application-independent (and quite obvious) requirement, analogous to the requirement that, in addition to their configuration-specific interconnections, all electronic components of a computer system (processor, monitor, printer, external hard drive, etc.) must also be plugged into the power network.

The main objective of SYNOPSIS application diagrams is to specify application-specific constraints among activities, expressed by flow and timing dependencies, in a more or less declarative fashion. Activity control ports are connected to dependencies only if the execution of those activities depends on other activities. To avoid the cluttering of SYNOPSIS diagrams with unnecessary connectors, the system enables designers to leave unconnected control ports of activities that do not depend on any other activities. The default semantics for such unconnected control ports are that the respective activities should be started as *early* as possible during an application run.

For example, in the File Viewer example application (Figure 3-1) we have left the Begin ports of activities Select File, Open DB and Start Viewer unconnected (in fact, we have also made them invisible). This implies the fact that, execution of those activities does not depend on any other activity, and should start immediately upon initiation of the application.

Our design decision to allow the existence of unconnected control ports in SYNOPSIS has the consequence that, even after all dependencies have been managed, some of the original and newly introduced modules might not be connected to a source of control. By default, these modules should start immediately upon initiation of the application. Therefore, before code can be generated, those modules have to be identified and "plugged into the power network", that is, connected by control flow dependencies to a source of control. Furthermore, in order to make the invocation of the application as

simple as possible, all application modules must be connected (by control flows) to a single *application entry point* (typically a double-clickable executable file).

One possible packaging strategy that achieves those goals is the following:

Step 1:
- Scan the application graph and find all source modules that are not connected to a source of control.
- Introduce a set of *packaging executable components*, one per host machine and per language for which unconnected source modules exist.
- Package calls to unconnected source modules inside the main program of the packaging executable corresponding to the host machine and language of each module.

This step introduces additional executable modules into the system. These executable modules must, in turn, be connected to a source of control, so that they can begin execution when the application starts.

Step 2:
- Find all executable programs that are not connected to a source of control.
- Introduce an *application entry executable component* into the system. This last component will be the entry point to the entire application, and can be written in an operating system script language, or in any other programming language.
- Package invocation statements for all unconnected executables inside the main program of the application entry executable.

The above algorithm introduces a set of additional activities and coordination processes into the application graph, as shown in Figure 5-12. It assumes that all languages used in the application permit the structuring of programs as application-executables-procedures hierarchies. Since this model is not true for all languages, the packaging strategy should be easily modifiable. At best, it should be a selectable parameter of the design process.

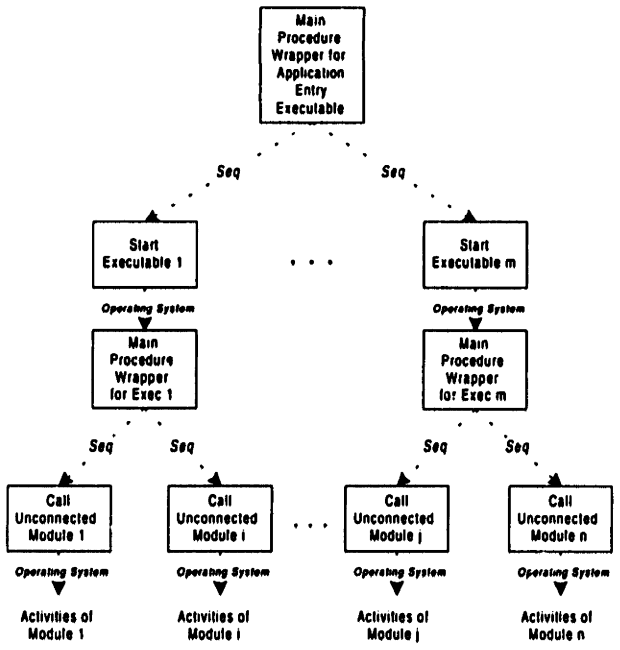


Figure 5-12: Additional activities introduced by the packaging algorithm.

b. Generate executable code

After the packaging step has been completed, SYNOPSIS graphs can be translated to sets of modules by a relatively straightforward process. Primitive coordination processes divide the graphs into families of partially ordered subgraphs, each corresponding to a sequential code block. Nodes in these subgraphs correspond to source code activities, that is, activities that are associated with source code procedures. Arcs in the graphs correspond to Seq and local variable primitive coordination processes that impose ordering constraints and data transfers through local variables.

The code generation process creates a topological sort of each subgraph (also handling the possibility of loops and branches), generates a procedure call statement for each activity in the subgraph, generates and automatically names local variables that transfer data among procedure call statements in the same subgraph, and packages all statements and variable definitions into sequential code blocks for the given language.

The code generator needs to be able to generate the correct syntax for each source language. Therefore, code generators need some language-specific knowledge for each language they should be able to handle. However, the language-specific knowledge required is quite limited. It basically consists of syntactic rules for generating procedure calls, declaring local variables, and generating procedure headers and footers. Support for a new language can thus easily be concentrated into a single class definition. Section 6.1.5 describes the requirements for supporting a new language in more detail.

5.3 An Algorithm for Integrating Activities and Dependencies

This section collects the techniques presented in Section 5.2 into a single algorithm for semi-automatically transforming SYNOPSIS architectural diagrams into executable applications. Stages 1, 3, and 4 of the algorithm are completely automatic. Stage 2 might require user input for selecting among multiple candidate processes or for inputting appropriate processes in cases where none can be found in the design element repository. Algorithm steps that require user input have been highlighted in bold typeface.

Generate_Application

Input: A SYNOPSIS diagram consisting of activities and dependencies
Output: A set of executable files implementing the target application

1. Decouple interface dependencies
2. Specialize generic design elements
3. Connect all modules to control
4. Generate executable code

Stage 1: ***Decouple interface dependencies***

Recursively scan all activities in the application graph.
For every activity associated with a code-level component,
    Scan all provided and expected interface definitions of the associated component.
        For every provided interface,
            Get the interface kind.
            If a caller activity has been defined for that interface kind,
                Check for "perfect match" special cases (see Section 5.2.3)
                If no "perfect match" interface is found at the other end,
                    Replace the original primitive activity with a composite pattern that includes a caller activity, as described in Figure 5-5.
        For every expected interface,
            Get the interface kind.
            If a wrapper activity has been defined for that interface kind,
                Check for "perfect match" special cases (see Section 5.2.3)
                If no "perfect match" interface is found at the other end,
                    Replace the original primitive activity with a composite pattern that contains a wrapper activity, as described in Figure 5-6.

Stage 2: ***Specialize generic design elements***

2-1 Scan graph and build a *to-do list* containing,
- all generic atomic activities (i.e. atomic activities not associated with a code-level component)
- all unmanaged dependencies

2-2 Repeat the following two operations until to-do list becomes empty,

a. Extract the next generic atomic activity.
For each executable specialization of that activity stored in the design repository,
Apply the compatibility checking algorithm of Figure 3-15.
If at least one matching executable specialization is found,
Ask user to select among them.
Otherwise,
Repeat while user input is invalid:
Ask user to provide a specialization for the activity.
Check validity of user-supplied activity (if must pass the compatibility checking algorithm and either be *atomic and executable* or *composite*)
Permanently store new activity in the repository.
Replace generic activity with selected or user-supplied specialization.
Apply Stage 1 of the algorithm to the replacing activity.
If replacing activity is composite and generic,
Scan activity decomposition and add all generic atomic activities and unmanaged dependencies found to the to-do list

b. Extract the next unmanaged dependency
For each coordination process[1] associated with a specialization of that dependency stored in the design repository,
Apply the compatibility checking algorithm of Figure 3-15.
If at least one matching coordination process is found,
Ask user to select among them.
Otherwise,
Repeat while user input is invalid:
Ask user to provide a compatible coordination process.
Check validity of user-supplied process (it must pass the compatibility checking algorithm and either be *atomic and executable* or *composite*)
Permanently store new process in the repository.
Manage dependency with the selected or user-supplied coordination process.
Apply Stage 1 of the algorithm to the managing coordination process.
If managing coordination process is composite,
Scan process decomposition and add all generic atomic activities and unmanaged dependencies found to the to-do list.

[1] The term coordination process here also includes atomic software connectors associated with executable dependencies.

Stage 3: ***Connect all modules to control***

3-1 a. Scan the application graph and find all source modules that are not connected to a source of control.
b. Introduce a set of *packaging* executable components, one per host machine and per language for which unconnected source modules exist.
c. Package calls to unconnected source modules inside the main program of the packaging executable corresponding to the host machine and language of each module.

3-2 a. Scan the application graph and find all executable programs that are not connected to a source of control.
b. Introduce an *application entry* executable component into the system.
c. Package invocation statements for all unconnected executables inside the main program of the application entry component.

Stage 4: ***Generate executable code***

4-1 Scan graph and divide into *sequential block subgraphs*[1].
For each subgraph,
Topologically order activities according to their sequentialization interdependencies.
Generate a call statement for each activity.
Generate a local variable declaration for each local variable coordination process.
Generate appropriate headers and footers for the enclosing sequential block.
Save resulting sequential block code into a file.

4-2 For each target executable:
Collect all source and object files of the executable[2].
Compile files and place resulting executable into target application directory.

[1] Sets of activities that will be packaged in the same sequential code block.
[2] These files are: (1) all source and object files referenced in the component descriptions of all activities to be included in the executable, and (2) all coordination code source files generated by Step 4-1 for the target executable.

5.4 Generating an Example Application

This section contains a step-by-step description of how the algorithm of Section 5.3 can generate an executable application by successively transforming a SYNOPSIS description of its architecture. Our example will be the File Viewer application that we used to illustrate the features of the SYNOPSIS language in Chapter 3 (see Figure 3-1).

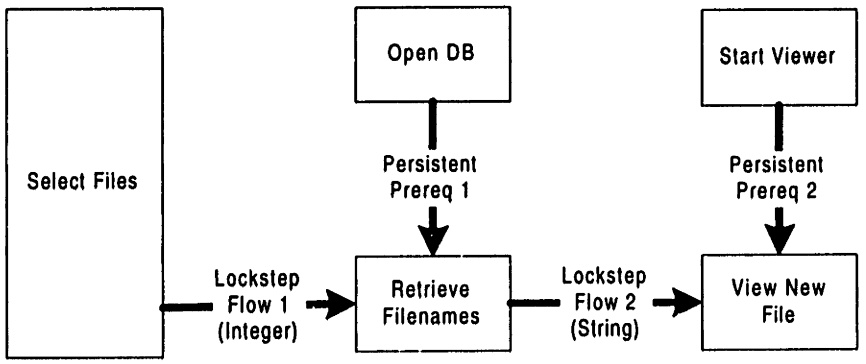


Figure 5-13: A simplified representation of the File Viewer example application (see Figure 3-1).

Figure 5-13 shows a simplified version of Figure 3-1 in which dependencies are represented by labeled thick solid arrows. Subsequent figures will show how this initial graph will be successively transformed by the various stages of the algorithm.

Stage 1: ***Decouple interface dependencies***

In this step, the algorithm scans all primitive activities and augments them with callers and wrappers as necessary.

Activity Select Files is associated with a source code procedure which expects the existence of another source code procedure with specified name and interface[1]. A procedure wrapper with appropriate attributes is inserted to handle this expectation (Figure 5-14(a)).

Activities Open DB and Retrieve Filenames are associated with simple source code procedures and do not expect any external components. Therefore, they need not be augmented.

[1] The definitions of provided and expected interfaces of the components associated with each of the five primitive activities of the File Viewer application can be found in Figure 3-4.

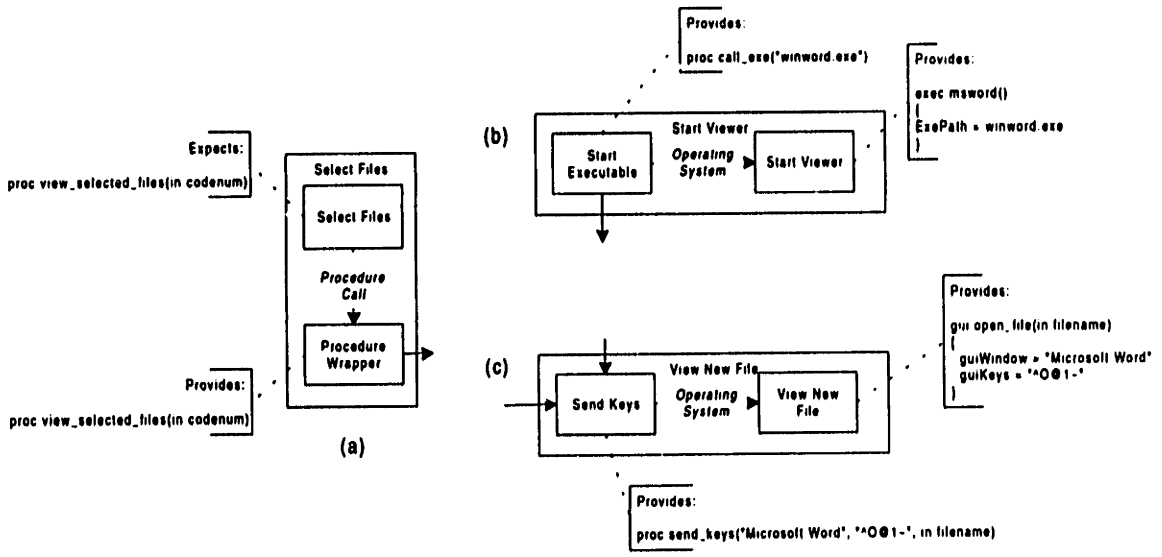


Figure 5-14: Augmentation of activities of File Viewer application by inserting caller and wrapper activities.

Activity Start Viewer is associated with an executable program. A caller activity that invokes the program is inserted (Figure 5-14(b)). Initially the caller activity is generic, because the system does not yet know in which language to generate the invocation statement. When that information later becomes available, the system will replace this activity with a language-specific executable specialization. The primitive coordination process labeled *Operating System*, simply states the fact that the operating system is responsible for actually transferring control from the invocation statement to the executable program.

Finally, activity View New File is a graphical-user-interface function, which is activated through a keystroke sequence. A caller activity that sends the appropriate keystroke sequence, and therefore activates the function, is inserted (Figure 5-14(c)). As before, the caller activity is initially a generic one, to be replaced by a language-specific executable specialization when more information becomes available.

After Stage 1 has been completed, the original application graph of Figure 5-13 has been transformed to the one shown in Figure 5-15. Primitive coordination processes are represented by dotted arrows.

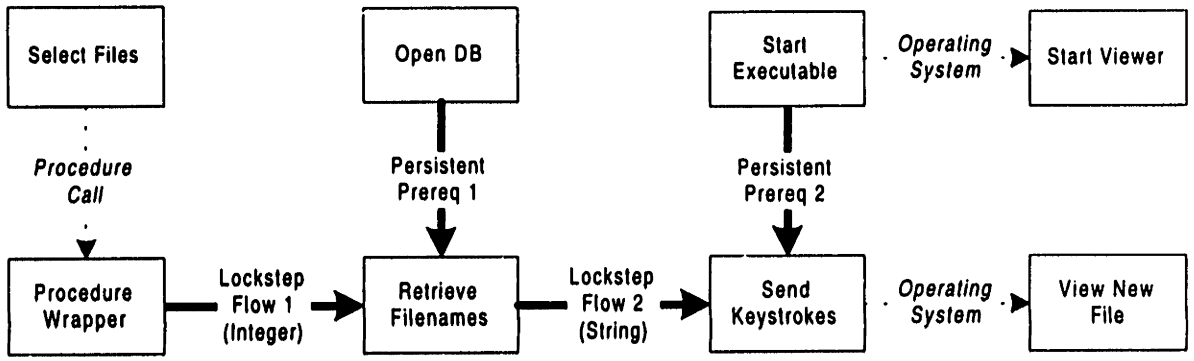


Figure 5-15: File Viewer application graph after completion of Stage 1.

Stage 2: Specialize generic elements

The graph of Figure 5-15 contains the following generic elements that must be either specialized or managed:

Generic activities	*Unmanaged dependencies*
Start Executable	Lockstep Flow 1
Send Keystrokes	Lockstep Flow 2
	Persistent Prereq 1
	Persistent Prereq 2

Stage 2 of the design algorithm successively handles each of the above elements.

Let us first look at the management of dependency Lockstep Flow 1. Chapter 4 contains a detailed discussion of coordination processes for flow dependencies. The algorithm would first manage the dependency with a generic flow coordination process, like the one shown in Figure 5-16(a). That process introduces additional dependencies and generic activities that must be recursively specialized. Chapter 4 describes a number of alternative ways of specializing the Transport Data activity. Those alternatives could have been organized in a repository of coordination processes as shown in Figure 5-17. Flow 1 specifies a flow between two procedures written in different languages (C and Visual Basic), which must be placed in different executable programs on the same host machine. Therefore, we must select a mechanism that is able to handle such cases. Microsoft Windows provides a protocol called Dynamic Data Exchange (DDE) to support such data exchanges. DDE transfer, however, only supports the exchange of string data, while in this flow we are transporting integer code numbers. The solution lies in specializing the management of the usability dependency in order to translate integers to and from strings (which act as an "interchange format" in this simple case, see Section 4.5.1). The final

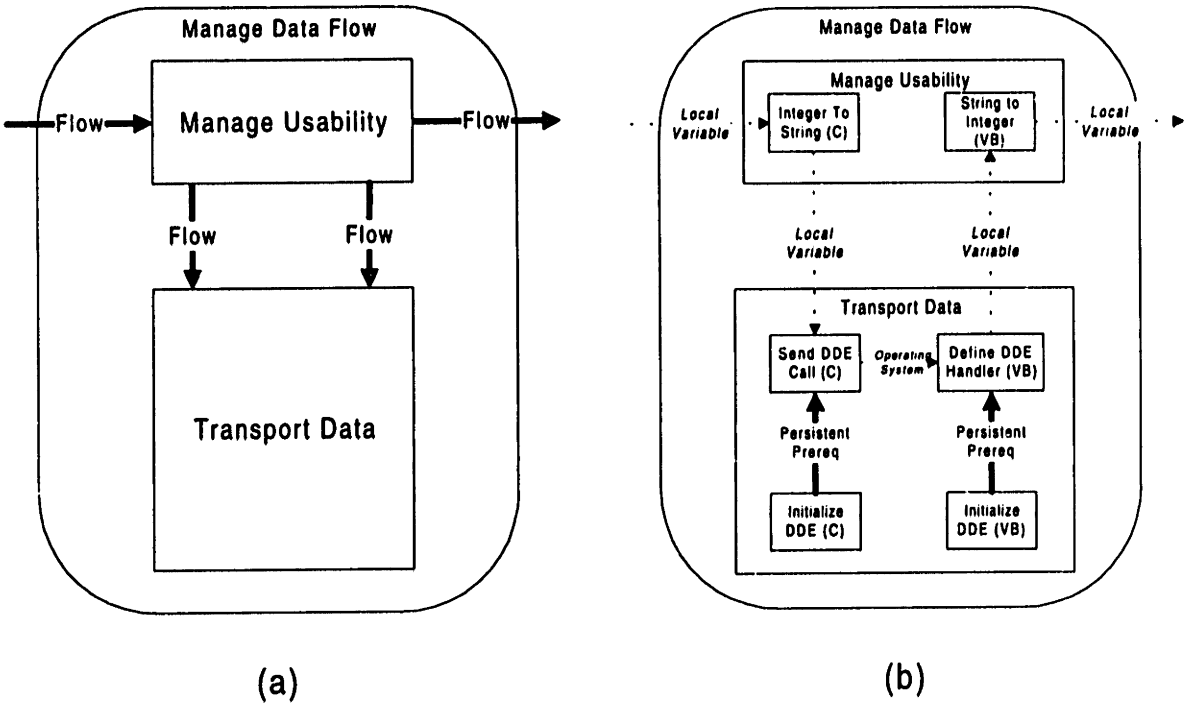


Figure 5-16: Generic and specialized processes for managing dependency Lockstep Flow 1.

decomposition of the coordination process for managing Flow 1 is shown in Figure 5-16(b).

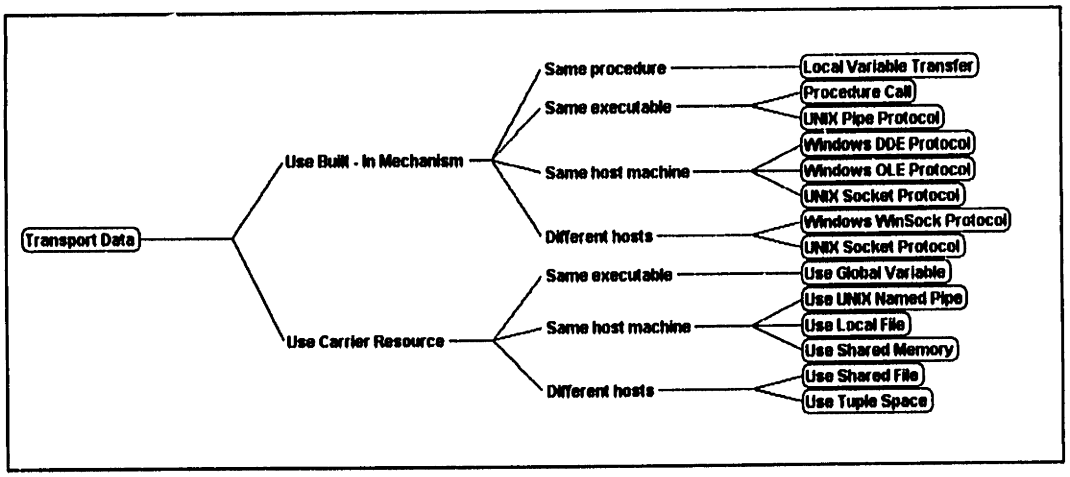


Figure 5-17: An excerpt of a coordination process library for managing data transport.

Figure 5-18 shows the File Viewer application graph after replacing Flow 1 with the coordination process of Figure 5-16(b).

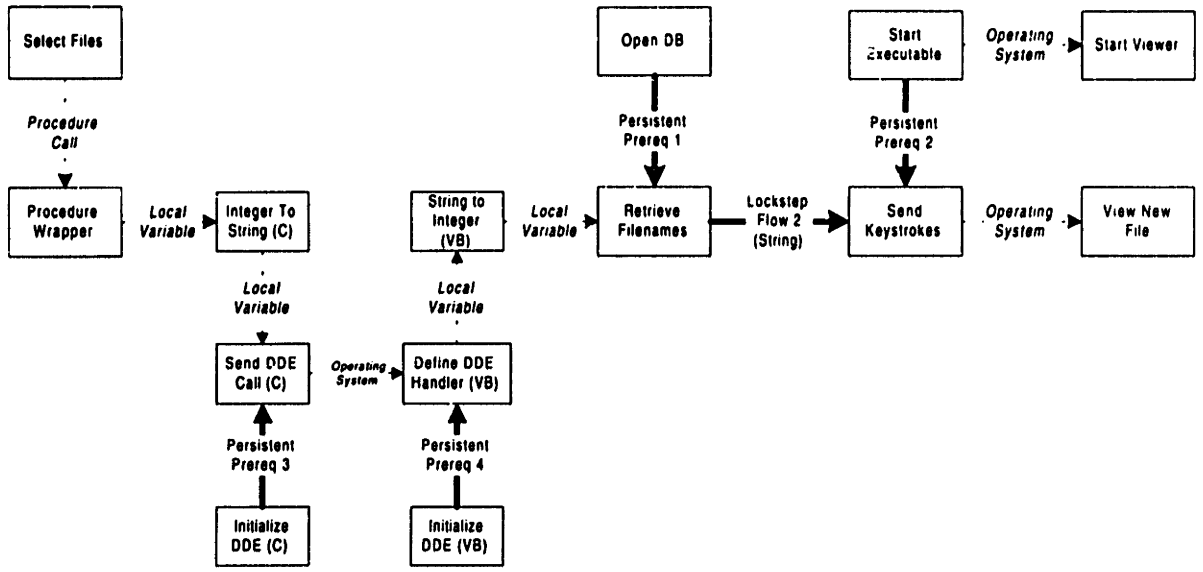


Figure 5-18: File Viewer application graph after managing dependency Lockstep Flow 1.

The remaining unmanaged dependencies can be managed in a similar fashion:

If we replace generic activity Send Keystrokes with its executable Visual Basic version, dependency Lockstep Flow 2 will be connecting two Visual Basic procedures. Therefore it can be trivially managed by packaging the calls to the two procedures inside another procedure and using local variables to transfer the data between them.

Figure 5-19 shows an excerpt from a repository of processes for managing persistent prerequisites. In this example, we choose to manage them simply by placing the precedent activities in the *initialization procedure* of their respective executable programs. The initialization procedure has the predefined name `Init_<language>` and is automatically placed before any other statement in the main procedure of the corresponding executable program by the code generator (Stage 4).

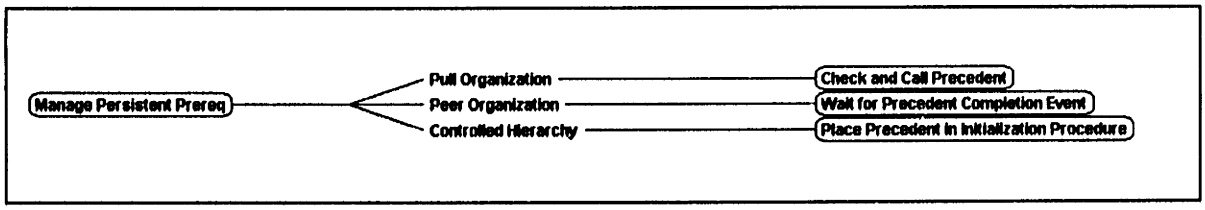


Figure 5-19: An excerpt of a coordination process library for managing persistent prerequisite dependencies.

Figure 5-20 shows the application graph after all above transformations have taken place. At this stage, all activities and dependencies have been replaced with executable specializations.

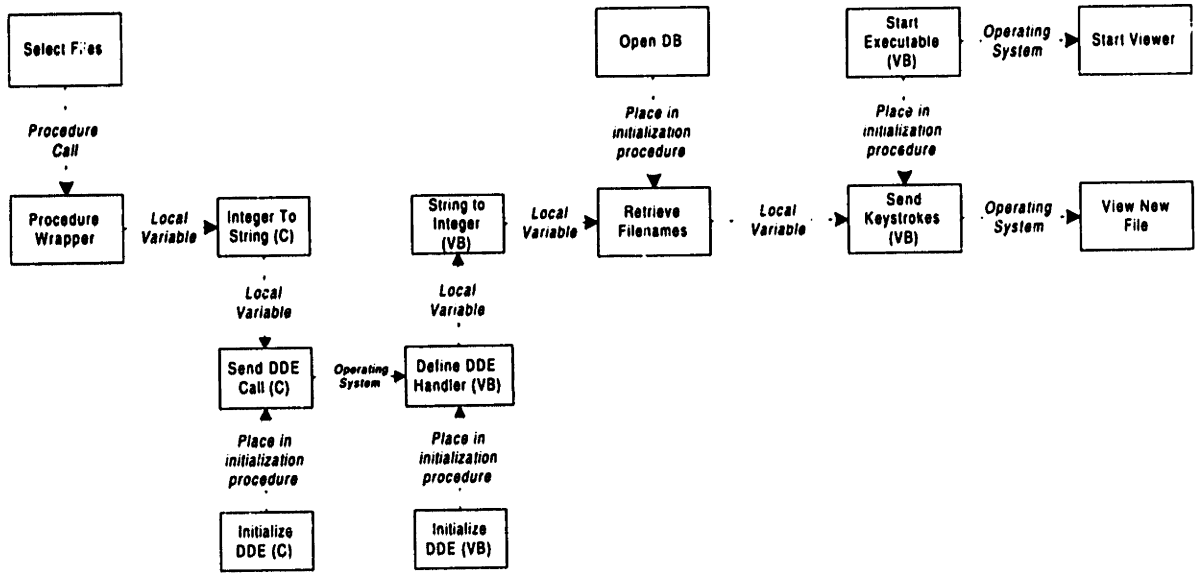


Figure 5-20: File Viewer application graph upon completion of Stage 2

Stage 3: Connect all modules to control

This stage begins by scanning all activities in order to detect the ones that do not receive control from anywhere else in the graph. In the diagram of Figure 5-20 such activities are the ones that have no arrow going into them. These activities are:

Activity Name	*Language*	*Comments*
Select Files	C	
Initialize DDE (C)	C	to be packaged inside initialization procedure of C executable
Initialize DDE (VB)	VB	to be packaged inside initialization procedure of Visual Basic executable
Open DB	VB	same as above
Start Executable (VB)	VB	same as above

One possible strategy for connecting those activities to control is the following:

A new executable program is created for each of the languages in which exist unconnected modules. In this application, unconnected source modules exist in two different languages: C and Visual Basic. Therefore, the system will create two new executables:

- executable `exe1.exe` for packaging C modules, and
- executable `exe2.exe` for packaging Visual Basic modules.

Each of the two executables will begin its execution from its main program. The main program will contain calls to:

- procedure `Init` (initialization procedure) for the given language
- other unconnected modules written in that language

In our example, the main procedure of `exe1`, written in C, packages calls to procedure `Init_C` and procedure `select_files`. The main procedure of `exe2`, written in Visual Basic, only packages a call to procedure `Init_VB`.

Finally, in order to be able to start the entire application from a single point, invocation statements for both executables will be packaged together inside the main program of yet another executable (`user.exe`) which will start the application.

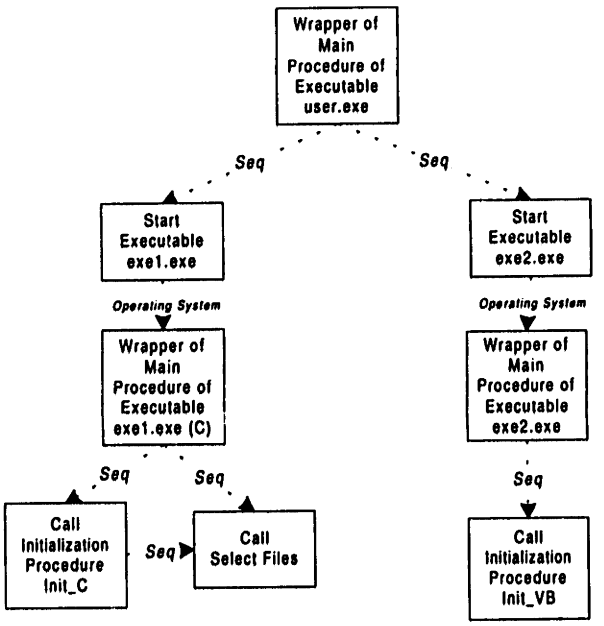


Figure 5-21: Packaging activities introduced by Stage 3.

The packaging activities introduced by this stage of the algorithm are shown in Figure 5-21. The resulting application graph after all the previous transformations have been applied is shown in Figure 5-22.

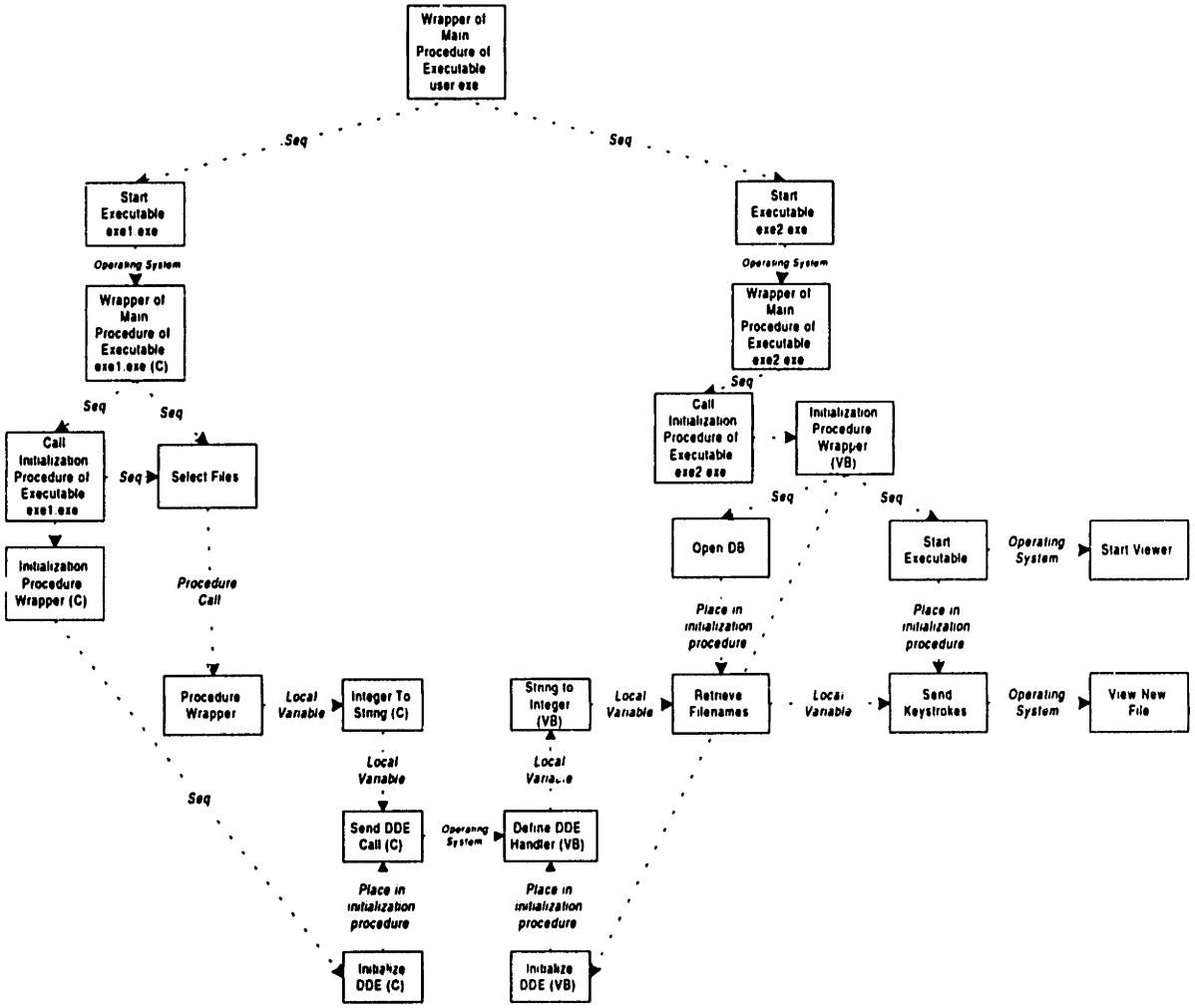


Figure 5-22: File Viewer application graph upon completion of Stage 3.

Stage 4: Generate code

The algorithm is now ready to transform the application graph into executable code. This is performed as follows:

- First, the application graph is separated into the subgraphs that represent sequential code blocks (C functions and Visual Basic procedures in this example). Activities of each subgraph are serialized according to their prerequisite and flow dependencies.

- Second, each subgraph is translated into its equivalent code block and saved into a file (Figure 5-23).

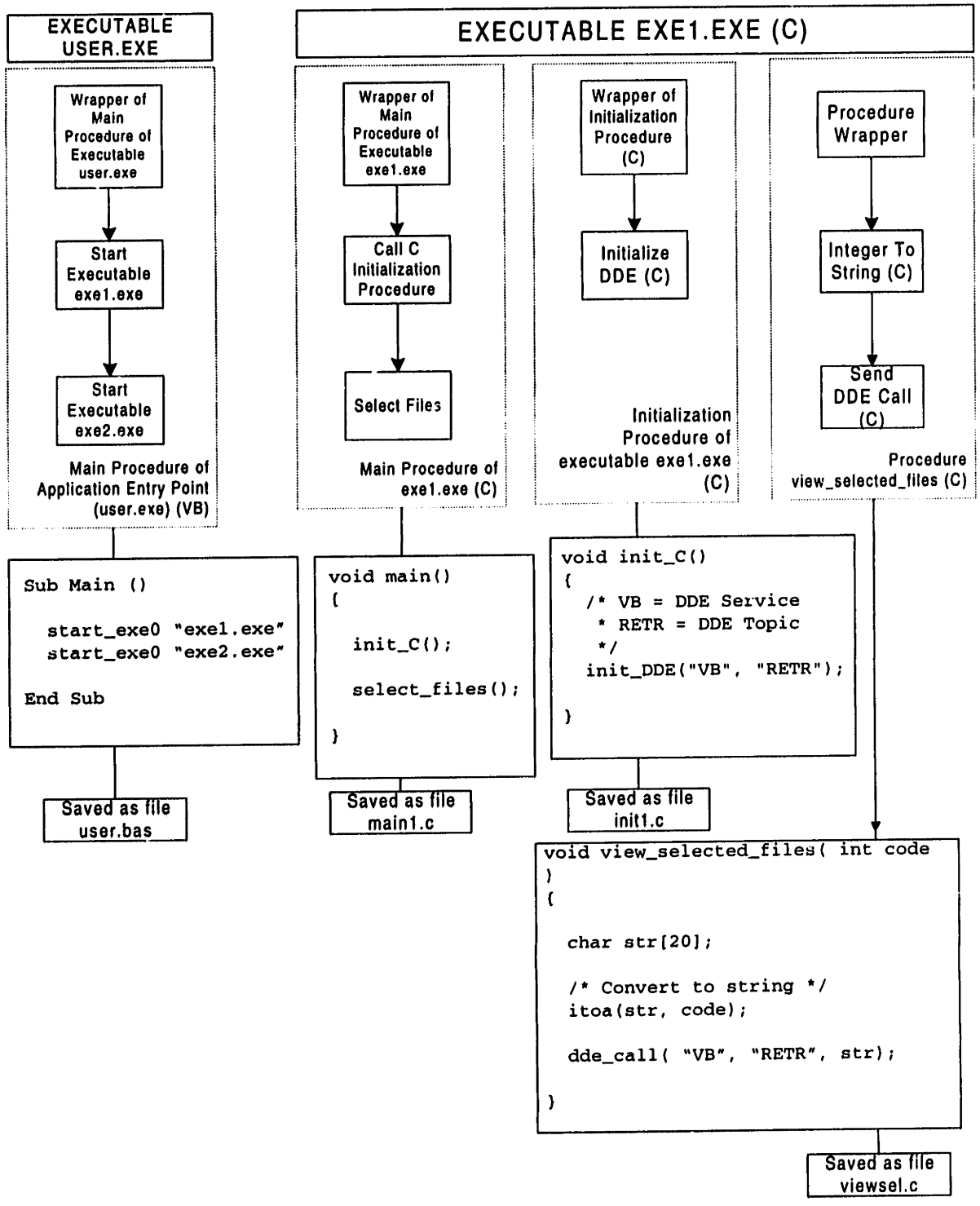


Figure 5-23 (a): Generating coordination code for the File Viewer application.

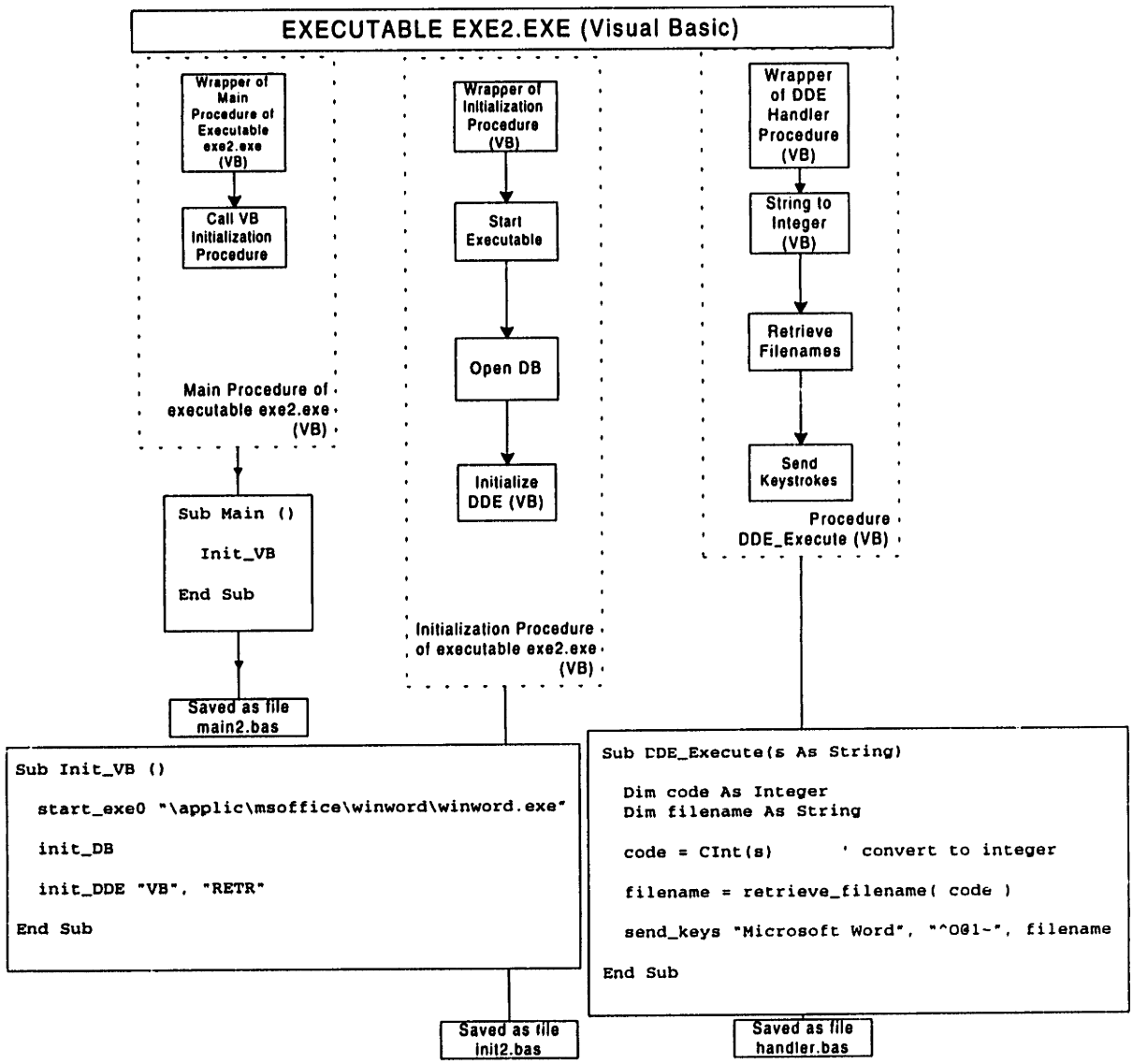


Figure 5-23 (b): Generating coordination code for the File Viewer application (continued).

- Third, the originally specified source and object files are collected together with the newly created coordination code files and compiled to generate each of the executables of the application (Figure 5-24).

The resulting set of executables can be invoked simply by double-clicking the executable file `user.exe`.

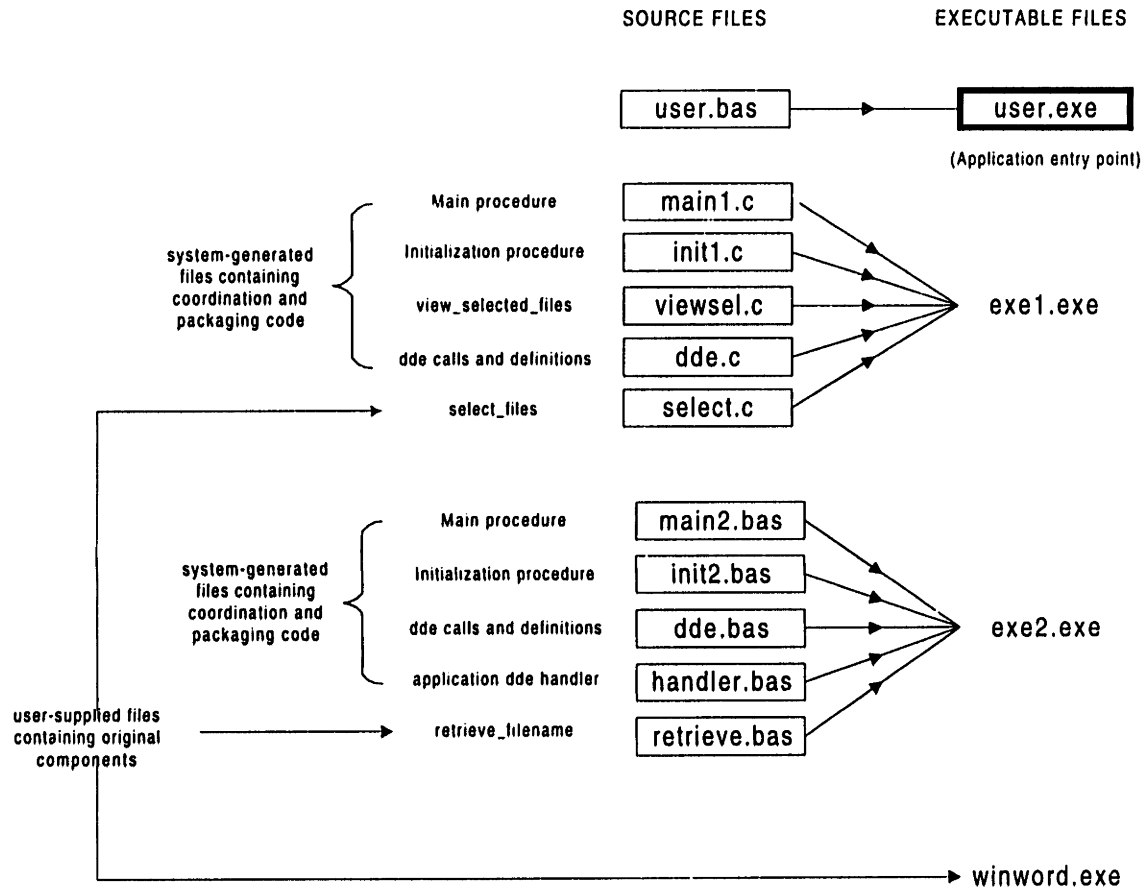


Figure 5-24: Final set of source and executable files for the File Viewer example application.

5.5 The Way Ahead

This chapter has concluded the description of the elements of the proposed process for developing component-based software applications. Our next two tasks are, to provide evidence for its feasibility and practical usefulness, and to compare it with related research efforts.

Chapter 6 begins with a description of SYNTHESIS, a prototype application development system that implements the ideas presented in Chapters 2-5 and demonstrates that they can indeed form the basis of a computer-assisted methodology for developing component based applications. The rest of the chapter describes our experiences from using SYNTHESIS to design four example applications.

Finally, Chapter 7 is devoted to a discussion of related research.

Chapter 6

Implementation and Experience

The previous chapters presented a process for developing software applications by combining existing components. The purpose of this chapter is to provide empirical evidence for the feasibility of the proposed approach, and explore different aspects of its usefulness in facilitating software reuse at both the code and architectural level. The chapter begins by presenting SYNTHESIS, a prototype implementation of the concepts and processes described in Chapters 2-5. SYNTHESIS has proven that the ideas presented in this thesis can indeed form the basis of a component-based software application development environment. The rest of the chapter is devoted to a detailed description and discussion of four experiments, performed using SYNTHESIS, that explore different aspects of the system.

6.1 SYNTHESIS: A Component-based Software Application Development Environment

SYNTHESIS[1] is a prototype implementation of the concepts and processes described in Chapters 2-5. It provides an integrated environment for developing software applications by combining existing components. The system provides support for:

- Creating and editing software architectural diagrams written in the SYNOPSIS language

[1] **syn·the·sis** (sin'the sis) *n.* [from Greek *syn-*, together + *thesis*, to place] **1** the putting together of parts or elements so as to form a whole **2** a whole made up of parts or elements put together **3** *Philos.* in Hegelian philosophy, the unified whole in which opposites (thesis and antithesis) are reconciled

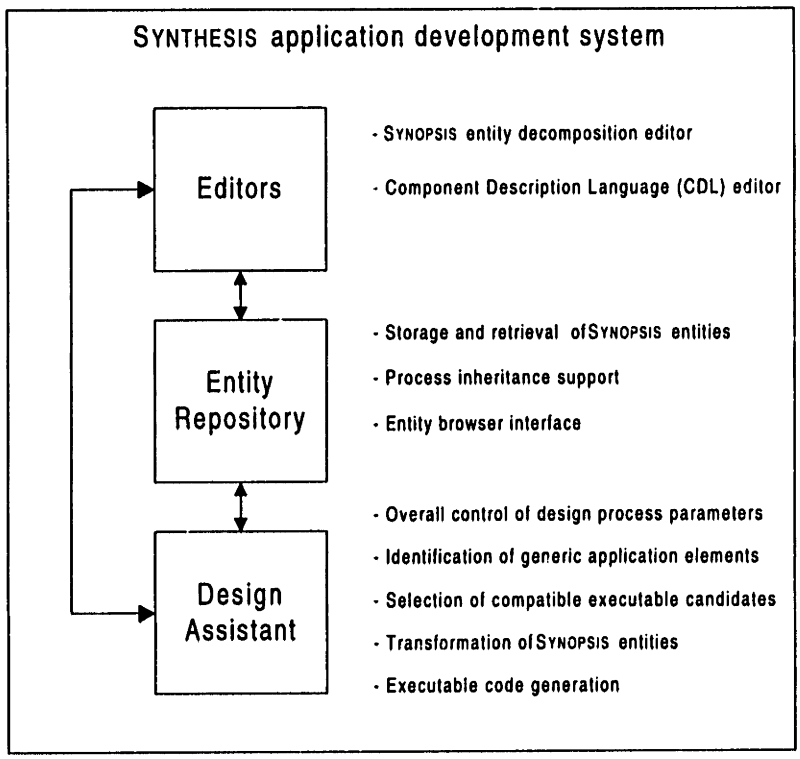


Figure 6-1: A high-level overview of the main pieces and functions of the SYNTHESIS prototype system.

- Maintaining repositories of SYNOPSIS entities (activities, dependencies, ports, resources), organized as specialization hierarchies.
- Transforming SYNOPSIS diagrams into executable code by applying the design process described in Section 5.3

The high-level structure of the SYNTHESIS system is shown in Figure 6-1.

6.1.1 Implementation Overview

The current implementation of SYNTHESIS runs under the Microsoft Windows 3.1 and Windows 95 operating systems. SYNTHESIS itself has been implemented by composing a set of components written in different languages.

- *Kappa-PC*, an object-oriented knowledge-based application development system [Intellicorp92], has been used for implementing the entity repository, as well as most of the functionality of the design assistant.

- *Visual Basic*, a rapid prototyping language [Microsoft93], has been used for implementing most of the user-interfaces, including the graphical editor driver.

- *VISIO*, a commercial application for drawing charts and diagrams [Shapeware94], has been used as the SYNOPSIS editor graphics front-end.

Kappa-PC provides an interpreted object-oriented programming language called KAL. It allows other applications to remotely invoke KAL statements using a Windows interapplication communication protocol called *Dynamic Data Exchange* (DDE). Likewise, Visual Basic supports the definition of subroutines that can be invoked from other applications using DDE. DDE has been used to implement the two-way communication between the Kappa-PC and Visual Basic parts of SYNTHESIS.

VISIO can act as a graphics server for other applications through a Windows protocol called *OLE automation* [Microsoft94]. OLE automation allows an application to act as a globally accessible object, offering a set of methods to other applications. Other applications can control the server by invoking the methods of its associated OLE object. OLE automation has been used to implement the communication between Visual Basic and VISIO.

6.1.2 SYNOPSIS Language Editors

SYNTHESIS supports all features of the SYNOPSIS architecture description language described in Chapter 3. It provides a graphical editor for creating and editing SYNOPSIS entities, implemented using the VISIO drawing tool. Composite entities can be *exploded* in place, in order to show their decomposition, and SYNOPSIS diagrams can be zoomed in and out to show lower-level elements in more detail. Exploding an atomic activity (i.e. an activity with no decomposition), brings up a component description language (CDL) editor for specifying the details of a code-level component to be associated with that activity (see Section 3.3.1.2).

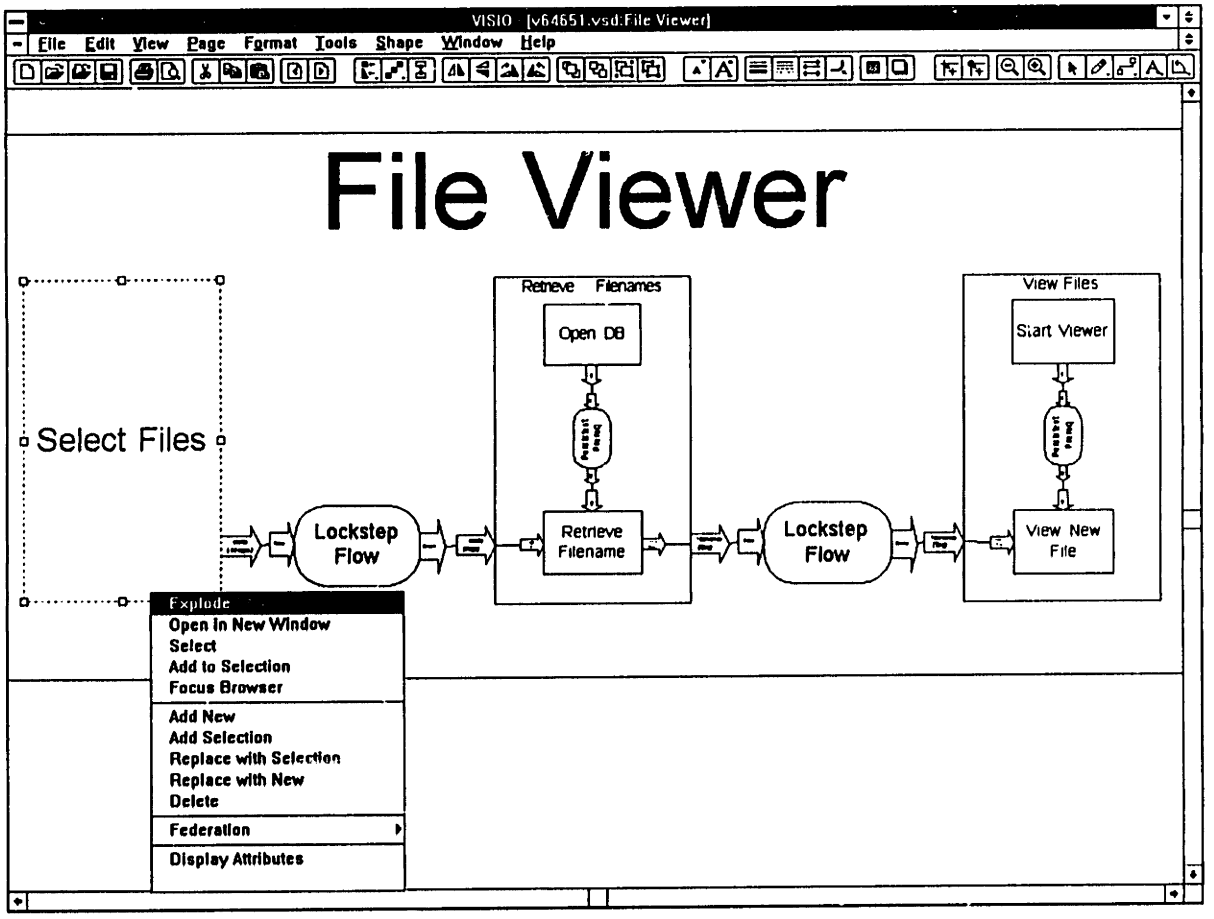


Figure 6-2: SYNOPSIS entity decomposition graphical editor.

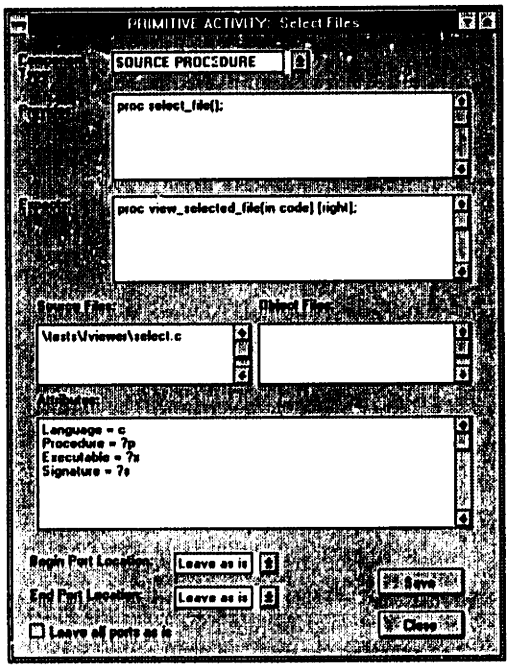


Figure 6-3: Component Description Language Editor

The currently supported code-level component types are listed in Table 6-1. As explained in Section 5.2.3, support for additional components can be built by specifying caller and wrapper activities for decoupling their interfaces into sets of independent local variables. SYNTHESIS allows designers to define a new component type, simply by adding an instance of a simple class, containing information such as the CDL keyword used to specify the component kind, and the names of caller and wrapper activities.

Component Type	*CDL Keyword*	*Description*
Source procedure	`proc`	A source code procedure or equivalent sequential code block (subroutine, function, etc.)
Source module	`module`	A source code module, consisting of one or more source code files and containing its own entry point (main program). Module components interact with the rest of the application through expected interfaces only.
Filter	`filter`	A source code procedure that reads its inputs from and/or writes its outputs to sequential byte streams
Executable	`exec`	An executable program
DDE server	`ddes`	A DDE server embedded inside an executable program
OLE server	`oles`	An OLE server embedded inside an executable program
Gui-Function	`gui`	A function provided by the graphical user interface of some executable program, typically activated through a key sequence

Table 6-1: Currently supported code-level component types

6.1.3 Entity Repositories

SYNTHESIS stores all SYNOPSIS entities created by users in specialization hierarchies, implemented using Kappa-PC's dynamic class hierarchies. All new entities are created as specialization children of some other stored entities. As described in Section 3.2, new entities *inherit* the decomposition and all attributes of their specialization parents and can differentiate themselves by subsequent modifications. SYNTHESIS supports *multiple inheritance*, that is, an entity can have more than one specialization parent. It that case, the entity inherits the union of the decomposition and attributes of all its parents.

Specialization hierarchies can be accessed and manipulated through an *entity browser* interface, providing tree-like views into the hierarchies (Figure 6-4). To facilitate browsing and navigation, specializations of a given entity can optionally be grouped together under a set of *bundles*. The entity browser distinguishes bundles from specialized entities, by enclosing the latter inside an oval. Browser elements without an oval are bundles.

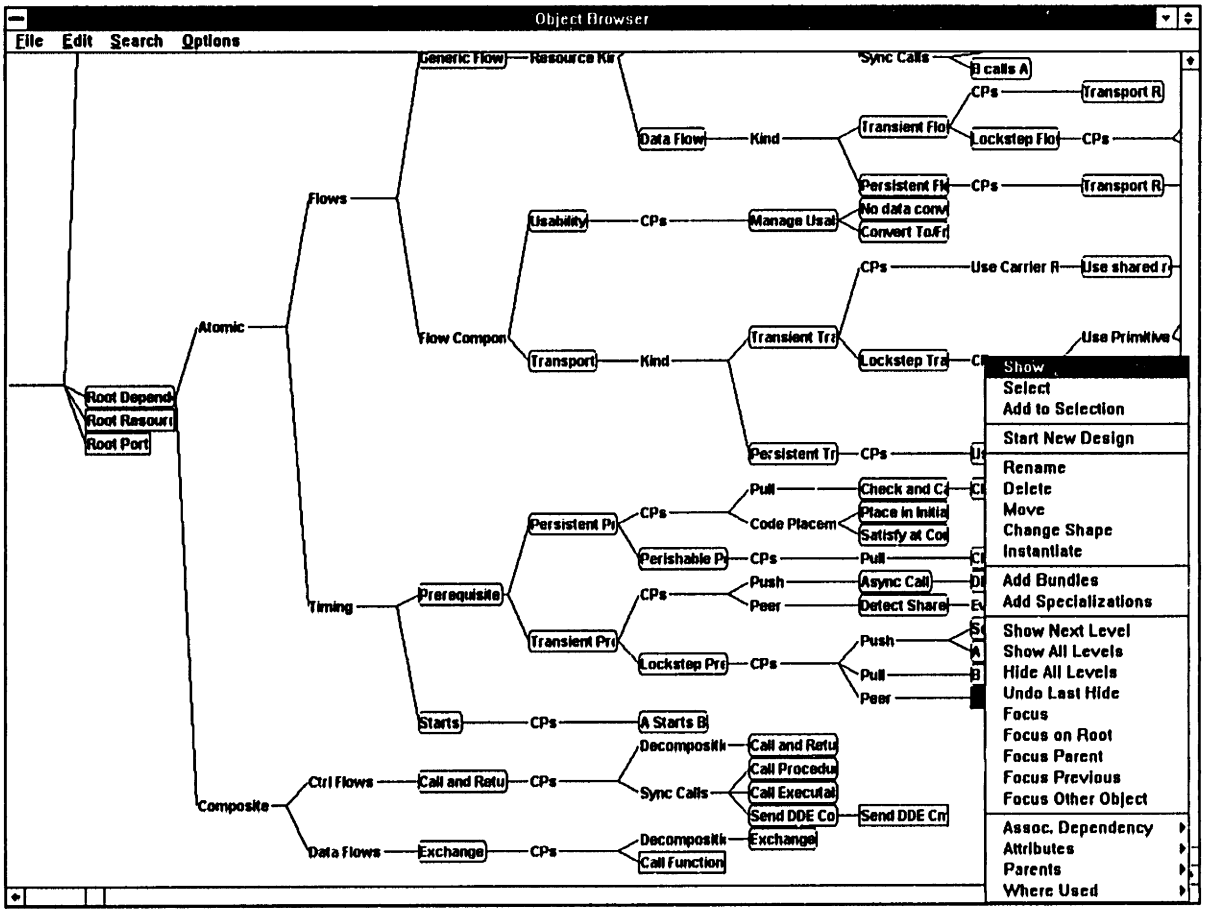


Figure 6-4: Entity browser interface showing part of the dependencies hierarchy. Dependencies close to the root are unmanaged. Dependency specializations with associated coordination processes are organized under bundles named "CPs".

The most important use of the entity repository in the current implementation is for maintaining a "design handbook" of increasingly specializaed dependency types. The "design handbook" is used by the design assistant, in order to semi-automate the process of generating executable applications, as described in Chapter 5.

Chapter 4 presented a general design space for dependencies and coordination processes. The current implementation of SYNTHESIS contains a repository based on translating a subset of that design space into a specialization hierarchy. The current repository contains coordination processes specialized for handling flow and timing dependencies among components written in C and Visual Basic, and running under UNIX or Windows.

It is emphasized that the entity repository can also be used for building specialization hierarchies of composite activities, representing alternative software architectures for solving frequently occurring problems. These architectures can be easily specialized and reused inside other activities using the SYNOPSIS machinery. In that manner, SYNTHESIS can potentially be useful as a tool for *architectural-level reuse*. A related project, which pursues similar concepts for the storage and reuse of business processes is described in [Malone93, Dellarocas94].

6.1.4 Design Assistant

The end-goal of SYNTHESIS is to assist users in transforming SYNOPSIS architectural diagrams into executable applications. Once a designer has completed entering and editing a SYNOPSIS application architecture (represented as a composite activity), he or she may enter the tool's *design mode.*

During design mode, the tool arranges its various windows as shown in Figure 6-5. The SYNOPSIS decomposition editor (in the lower half of the screen) displays the current state of the architectural diagram and updates it automatically whenever a new transformation (replacement of activity or management of dependency) takes place. The entity browser (in the upper left part of the screen) is used for displaying and selecting compatible specializations for application design elements. Finally, a new window, the *design manager* (in the upper right part of the screen), summarizes the status of the design process and allows users to control its parameters.

During the first stage of the design process, the design assistant scans the target application architecture, augments primitive activities with callers and wrappers (as described in Section 5.2.3), and collects all generic elements into a *to-do list* displayed in the design manager window.

The design process (see Section 5.3) then proceeds as follows:

The design assistant picks the next element of the to-do list. It zooms in that element in the decomposition window and scans all corresponding specializations that exist in the repository. For each of them, it optionally applies the compatibility checking algorithm of Figure 3-15, in order to determine whether it can be used in the current context. Finally, it presents the results of that search in the entity browser window (with compatible elements highlighted) and prompts the user to select one of the candidates (Figure 6-5). If no compatible candidate could be found, it prompts the user to enter a compatible specialization of the corresponding element and continue the process. This process continues until the to-do list becomes empty. The system then automatically generates code, as explained in Section 5.2.4.

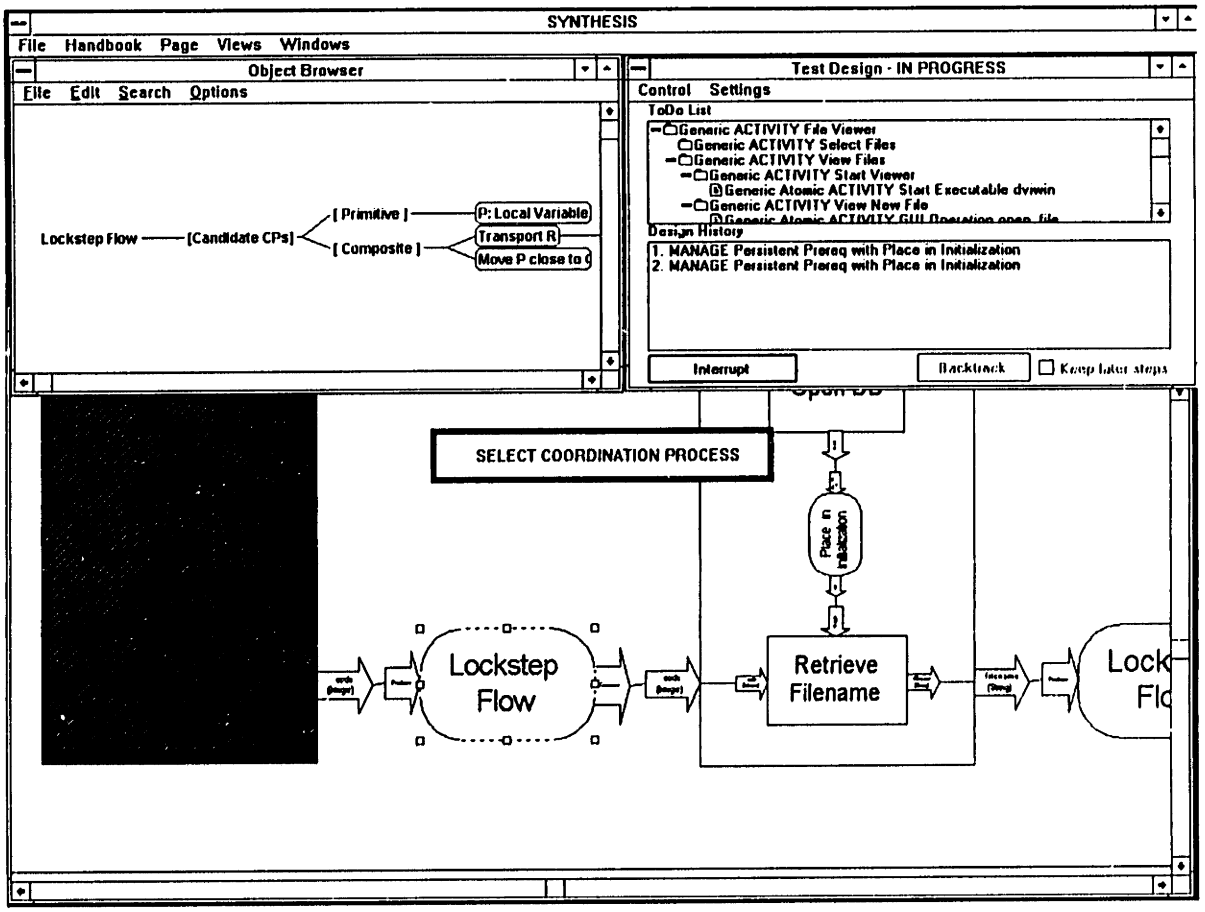


Figure 6-5: Configuration of SYNTHESIS windows during design mode.

The design process can be customized in a variety of ways.

- Designers can manually select each new element to be managed, rather than follow the ordering of the to-do list.

- Designers can input an *evaluation function* that helps the system perform an automatic ranking of compatible candidates. The system can then perform candidate selection completely automatically and only ask for user input whenever there are problems (e.g. no candidates or more than one candidates with the same score).

- Successive transformations of the original application diagram (stored as composite activities) can optionally be stored as successive specializations in the activity hierarchy. The system can thus keep a *design history*, which allows designers to easily backtrack to a previous stage of the design and choose a different design path. In that manner, exploratory design and maintenance of alternative implementations can be facilitated.

The design assistant is currently performing a simple exhaustive search of all candidate elements for a given transformation. For large repositories of processes, this may be

prohibitively expensive. Additional filtering and pruning methods (e.g. based on attribute values) should be developed to speed up the search in such cases.

6.1.5 Code Generation

The code generation component is currently capable of generating Visual Basic and C code only. However, as explained in Section 5.2.4, most of the code generation machinery is language independent. SYNTHESIS allows designers to add support for additional languages simply by defining a new instance of a class containing a few relatively simple language-specific methods (Table 6-2).

Method	*Arguments*	*Description*
`MakeArgumentDecl`	name, type	Generate a declaration for a procedure[1] formal parameter
`MakeCall`	name, type, parameter list	Package a name and list of actual parameters into the correct syntax for a local procedure call
`MakeCondBranch`	condition list	Generate the correct language-specific syntax for a conditional branch
`MakeFileFooter`	name	(Optional) Generate language-specific source code file footer statements
`MakeFileHeader`	name	(Optional) General language-specific source code file header statements
`MakeProcFooter`	name	Generate the syntax for ending a procedure call of given name
`MakeProcHeader`	name, type, parameter list	Generate a header statement for a procedure of given name, type, and formal parameter list
`MakeLabel`	label number	Generate a label (branch target)
`MakeLocalDecl`	name, type	Generate a local variable declaration of given name and type

Table 6-2: Set of methods that must be defined in order to be able to generate code for a new language.

[1] The term "procedure" should be replaced with the appropriate name for sequential code blocks in the target language.

6.2 Experiment 1: A File Viewer Application

6.2.1 Introduction and Objectives

The purpose of our first experiment was to investigate how well our system can integrate a set of heterogeneous components in a relatively simple application. We were also interested in testing the *exploratory design* capabilities of our approach. More specifically, our objective was to demonstrate how the system can assist designers integrate the same set of components in a number of different ways, by selecting alternative coordination processes for managing the specified dependencies.

The target application is a simple file viewer, whose purpose is to repeatedly ask users for code numbers through a simple user interface, retrieve filenames corresponding to each user-supplied code number from a database, and display the contents of the retrieved filenames using a text editor. The file viewer application has already been introduced as a vehicle for explaining various aspects of the system in Chapters 3 and 5.

The components used were source modules written in different programming languages (C and Visual Basic), as well as one executable program.

6.2.2 Description of the Experiment

Figure 6-6 depicts a SYNOPSIS architectural diagram for the File Viewer application. The diagram is explained in detail in Section 3.2, where it was used to give an overview of the SYNOPSIS language. The components used to implement each of the activities shown in the architectural diagram were the following:

- a C source code module, used to implement activity Select Files
- a Visual Basic source code module, used to implement activities Open DB and Retrieve Filename
- a commercial text editor in executable form, used to implement activities Start Viewer and View New File

Figure 6-7 depicts the SYNOPSIS definitions linking executable activities to their associated implementation-level components.

Section 5.4 gives a detailed, step-by-step account of how SYNTHESIS can generate an executable implementation of an integrated file viewer application by successively transforming the architectural diagram of Figure 6-6. The generation process is based on successively replacing unmanaged dependencies with specializations associated with a compatible coordination process.

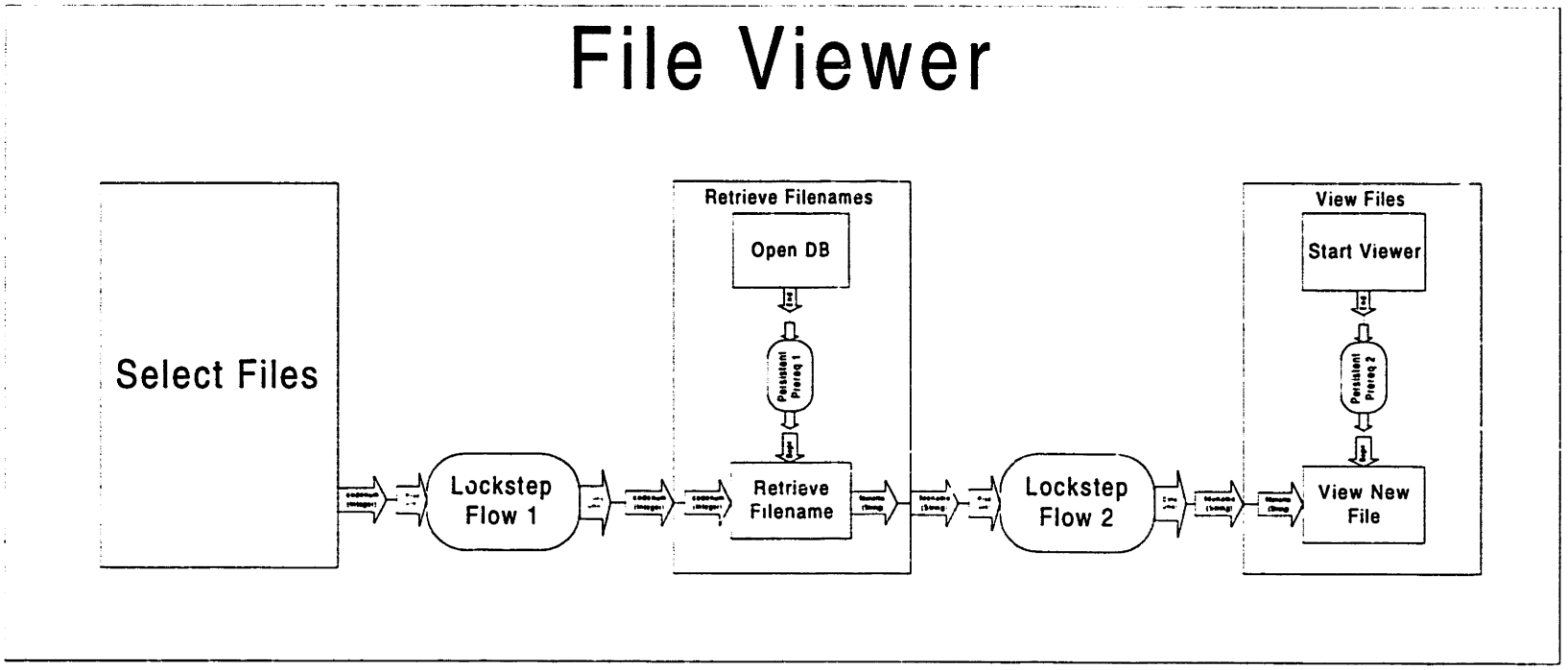


Figure 6-6: SYNOPSIS architectural description of a File Viewer application (Repeated from Figure 3-1).

```
Component "Select Files" IsA Procedure

Provides:
    proc select_file();

Expects:
    proc view_selected_file(in codenum:Integer);

Source Files:
    \fviewer\select.c

Attributes:
    Language = c
```

```
Component "Retrieve Filename" IsA Function

Provides:
    func retrieve_filename(in codenum:Integer):String;

Source Files:
    \fviewer\retrieve.bas

Attributes:
    Language = vb
```

```
Component "Open DB" IsA Procedure

Provides:
    proc init_DB();

Source Files:
    \fviewer\retrieve.bas

Attributes:
    Language = vb
```

```
Component "Start Viewer" IsA Executable

Provides:
    exec msword();

Attributes:
    ExePath = \applic\msoffice\winword\winword.exe
```

```
Component "View New File" IsA Gui-Function

Provides:
    gui open_file(in filename:String);

Attributes:
    guiWindow = "Microsoft Word"
    guiKeys = "^O@1~"
```

Figure 6-7: Component descriptions for the File Viewer example application (Repeated from Figure 3-4).

In several cases, more than one compatible specializations might exist for a given dependency. In those cases, users can eliminate some alternatives by using common sense or design heuristics built into the system. If the remaining alternatives are still more than one, users can explore the use of more than one processes, thus creating several implementations of the same application. They can then compare their run-time efficiency, or other performance criteria, and select the one that performs best.

In the file viewer application, there were two cases where more than one compatible coordination processes could have been selected:

When managing Persistent Prerequisites 1 and 2, there were three compatible choices (Figure 5-18):

- Place precedents in the initialization procedure of the same executable as the consequent
- Use a "pull" protocol: before each invocation of the consequent, check if the precedent has occurred. If it has not, call the precedent before passing control to the consequent.
- Assign the precedent to an independent thread, which generates an event when the precedent completes. Before each invocation of the consequent, wait until the completion event occurs.

Of the previous three processes, choice 3 is clearly too heavyweight for the application at hand. However, choices 1 and 2 could be equally efficient.

Likewise, when managing Flow 1, there were two compatible choices (Figure 5-17):

- Use a DDE protocol
- Use a shared file for transmitting data and a separate synchronization protocol for managing the embedded lockstep prerequisite. Selection of a process for managing the lockstep prerequisite results in more choices, such as using semaphores, using a push or pull organization based on a procedure call, etc.

Choice 1, based on operating system support for transferring data and control between two different executable programs residing in the same Windows-based machine, is more efficient when transmitting small pieces of data, like we do in this example. However, for large data transfers, choice 2 might also become a viable alternative.

In the implementation described in Section 5.4, we have selected choice 1 for managing each of the above dependencies. We also tested the use of choice 2 for both the prerequisites and the flow dependency. Appendix A.1 lists the coordination code generated by SYNTHESIS in each of the two cases.

6.2.3 Discussion

This experiment has demonstrated that SYNTHESIS can resolve low-level problems of interoperability, such as incompatibilities in programming languages, data types, procedure names, and control flow paradigms. The components used in this example were written in different languages and forms (source and executable). However, their interfaces were essentially compatible. For example, the component to which activity Select Files expects to send each user-supplied code number (a procedure call accepting one argument), was essentially compatible with that provided by activity Retrieve Filename (a procedure call accepting one argument). The mismatches were all of a low-level nature (different procedure names, different languages, different executables), and SYNTHESIS was able to resolve them completely automatically. The next experiment will focus on components which have some more fundamental *architectural mismatches* in their interfaces and assumptions.

The experiment has also demonstrated the exploratory mode of design encouraged by the system. The system is able to use the compatibility checking mechanism described in Section 3.4 to automatically eliminate some of the candidate coordination processes for a given dependency. For example, when managing Flow 1, it was able to eliminate all processes of Figure 5-17 which were not suitable for managing flows between Windows-based components that will be packaged in different executables. However, even after this elimination has taken place, more than one compatible alternative processes typically remained. Currently, designers are responsible for selecting among alternative processes by using their experience and common sense judgment. A promising path of future research is to codify design rules for selecting coordination processes. The development of successful design rules will enable the complete automation of application generation from SYNOPSIS diagrams.

6.3 Experiment 2: Key Word In Context

6.3.1 Introduction and Objectives

In an influential 1972 paper [Parnas72], David Parnas described the *Key Word in Context* (KWIC) index system and used it to compare different approaches for dividing systems into modules. Recent papers on software architecture [Garlan88, Garlan94] have used the same application to illustrate and contrast different software architectural styles.

Parnas described his Key Word in Context (KWIC) example application as follows:

> The KWIC [Key Word in Context] index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

In this experiment, we made yet another use of Parnas's KWIC example. Our objectives were the following:

- Test the claim that SYNTHESIS can help generate applications from sets of independently selected components with different *architectural assumptions*, that is, assumptions about the structure of the application in which they will be used.

- Investigate the extent to which the same set of components can be integrated in a variety of different overall architectures, by selecting different coordination mechanisms.

6.3.2 Description of the Experiment

It is easy to construct a SYNOPSIS architectural diagram from Parnas's description of the KWIC system (Figure 6-8). In this experiment we used three sets of alternative component implementations:

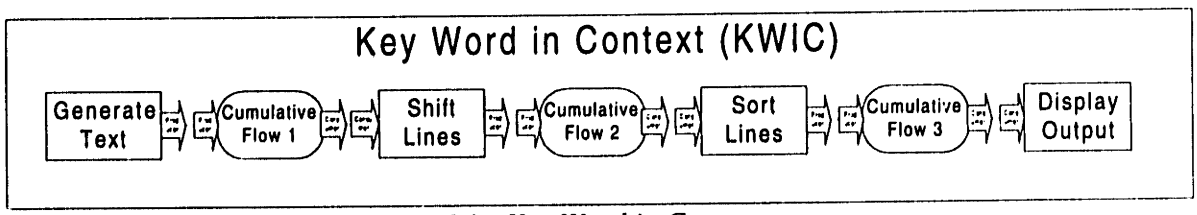
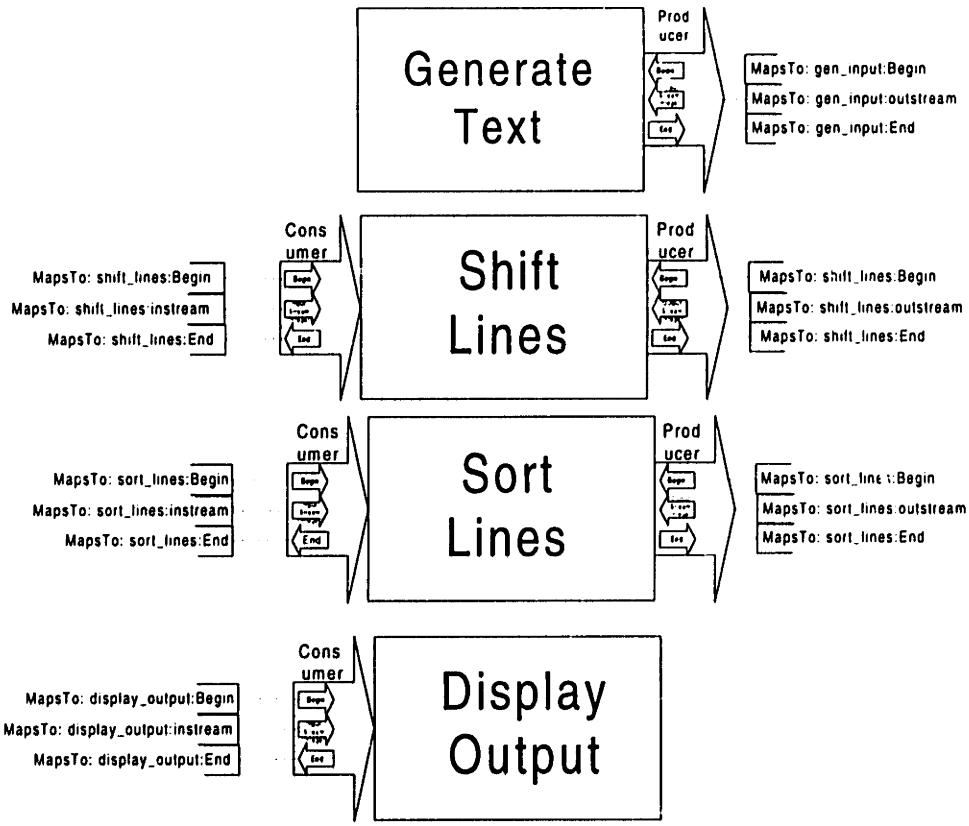


Figure 6-8: SYNOPSIS description of the Key Word in Context system.



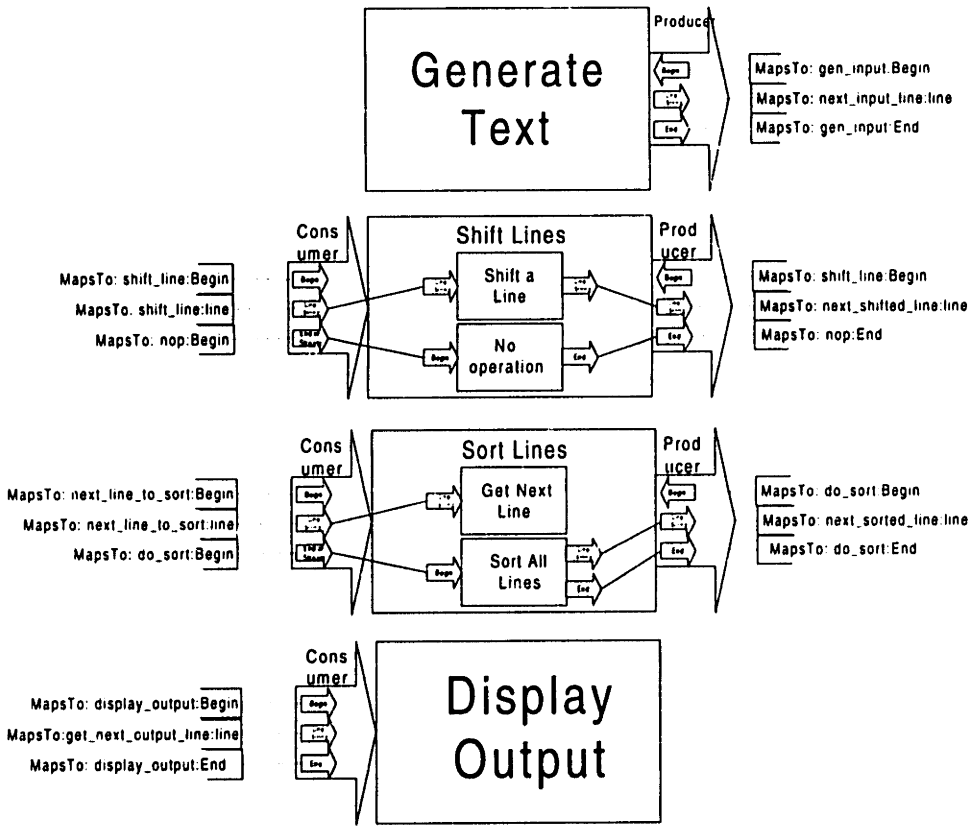

```
Component "Generate Input" IsA Filter

Provides:
  proc gen_input(in outstream:Integer);
Source Files:
  /kwic/filter/kwic.c
Attributes:
  Language = c
```

```
Component "Shift Lines" IsA Filter

Provides:
    proc shift_lines(in instream:Integer,
                     in outstream:Integer);
Source Files:
      /kwic/filter/kwic.c

Attributes:
    Language = c
```

```
Component "Sort Lines" IsA Filter

Provides:
    proc sort_lines(in instream:Integer,
                    in outstream:Integer);
Source Files:
    /kwic/filter/kwic.c
Attributes:
    Language = c
```

```
Component "Display Output" IsA Filter

Provides:
    proc display_output(in instream:Integer);

Source Files:
    /kwic/filter/kwic.c

Attributes:
    Language = c
```

Figure 6-9: SYNOFSIS definitions for Set A of KWIC components (filter implementations)




```
Component "Generate Input" IsA Procedure

Provides:
    proc gen_input();
Expects:
    proc next_input_line(in line:String);
Source Files:
    /kwic/cs/kwic.c
Attributes:
    Language = c
```

```
Component "Get Next Line" IsA Procedure

Provides:
    proc next_line_to_sort(in line:String);
Source Files:
    /kwic/cs/kwic.c
Attributes:
    Language = c
```

```
Component "Shift a Line" IsA Procedure

Provides:
    proc shift_line(in line:String);
Expects:
    proc next_shifted_line(in line:String);
Source Files:
    /kwic/cs/kwic.c
Attributes:
    Language = c
```

```
Component "Sort All Lines" IsA Procedure

Provides:
    proc do_sort();
Expects:
    proc next_sorted_line(in line:String);
Source Files:
    /kwic/cs/kwic.c
Attributes:
    Language = c
```

```
Component "Display Output" IsA Procedure

Provides:
    proc display_output();
Expects:
    func get_next_output_line():String;
Source Files:
    /kwic/cs/kwic.c
Attributes:
    Language = c
```

Figure 6-10: SYNOPSIS definitions for Set B of KWIC components (server implementations)

- Set A consists of components implemented as *UNIX filters*. UNIX filters receive their inputs and produce their outputs as sequential byte streams, to be read and written from (to) a pipe or file. Figure 6-9 lists the SYNOPSIS definitions of Set A.

- Set B consists of components implemented as source code "servers". Each component provides a "server" procedure interface, through which inputs can be fed by some other component. The component itself expects the existence of another "server" procedure, which it repeatedly calls to pass each of its output values. In contrast to Set A, where input and output takes place at the level of sequential byte streams, Set B components communicate at the level of lines of text (newline-terminated strings). Figure 6-10 lists the SYNOPSIS definitions of Set B.

- Set C consists of a mixture of components from the previous two sets. More specifically, in Set C, activities Generate Text and Sort Lines (1st and 3rd) are implemented using the corresponding filter components from Set A, while activities Shift Lines and Display Output (2nd and 4th) are implemented using the corresponding source code server components from Set B.

In [Garlan94], Garlan and Shaw examine three different architectures for implementing the KWIC system[1]:

- pipes and filters
- main program/subroutine with shared data
- implicit invocation

In this experiment, we used the SYNTHESIS system to generate an implementation of the KWIC example for each of the above three sets of components and each of the three different architectures. Overall, our objective was to construct 9 alternative implementations.

SYNTHESIS was able to generate all 9 implementations. In 7 out of 9 cases the results were of comparable efficiency with hand-written code. In the two cases where the system was not able to generate an efficient solution, the components used could not efficiently be integrated in the desired organization because of conflicting assumptions embedded in their code. The results of the experiment are summarized in Table 6-3. Appendix A.2 contains the coordination code generated by the SYNTHESIS in each case.

The following is a description of the most interesting aspects of each experiment:

[1] Garlan and Shaw also examined a fourth solution, based on abstract data types. However, this solution represented a different implementation of the components, rather than a different architecture. The organization of components in their abstract data type solution resulted in a main program-subroutine architecture.

	Components	*Architecture*	*Lines of automatically generated coordination code **	*Lines of manually written coordination code **
1	Set A (Filters)	Pipes	34	0
2	Set A (Filters)	Main Program/Subroutine	30	0
3	Set A (Filters)	Implicit Invocation	150	0
4	Set B (Servers)	Pipes	78	16
5	Set B (Servers)	Main Program/Subroutine	35	0
6	Set B (Servers)	Implicit Invocation	95	0
7	Set C (Mixed)	Pipes	66	0**
8	Set C (Mixed)	Main Program/Subroutine	56	0**
9	Set C (Mixed)	Implicit Invocation	131	16

* Line count does not include blank lines or comments.

** Makes use of manually written code for implementation 4.

Table 6-3: Summary of the Key Word in Context experiments

Set A: Filter Components

Implementation 1: Pipe architecture

Pipe coordination processes are the most natural mechanism for integrating filter components. Filters are examples of components that embed part of the support for managing their associated flows into their code. Their abstract producer and consumer interfaces decompose into identifiers of the carrier resource (a byte stream resource, such as a pipe or file). The actual statements for writing and reading data into (from) the carrier resource are embedded inside the components. Coordination processes for managing flows between two filter components can be derived from the generic data flow coordination model presented in Figure 4-21 by removing the Write and Read activities, which are now part of the component. It is also assumed that there are no usability considerations. The resulting generic process is shown in Figure 6-11(b) and forms the basis for a set of coordination mechanisms for managing data flows among filters.

The specialization that uses UNIX pipes as the carrier resource is shown in Figure 6-11(c) Since a UNIX pipe is only accessible to processes created by a common ancestor, the process manages the accessibility dependency for the pipe by forking a child process to execute the data flow consumer activity. The data flow producer activity is executed by the same thread that opens each pipe.

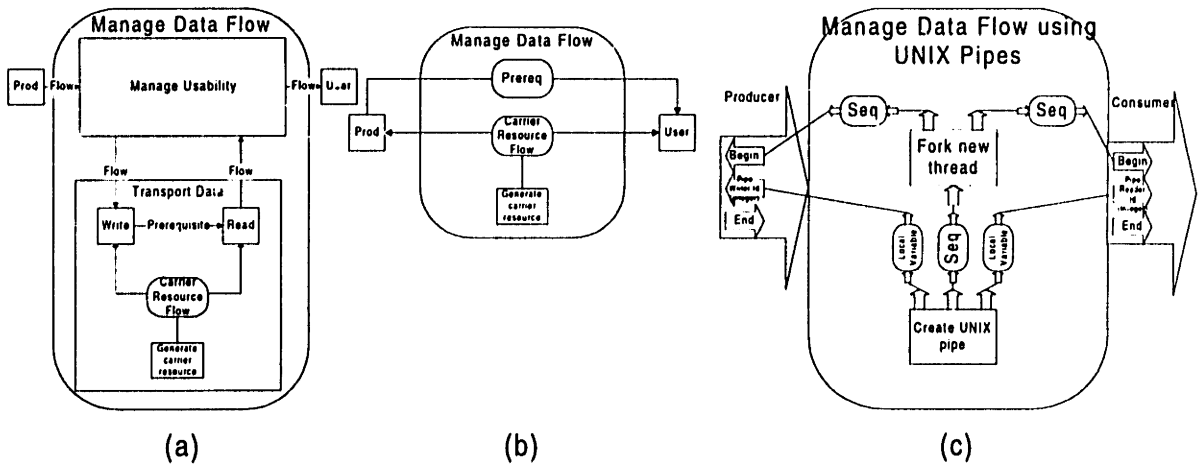


Figure 6-11: A coordination process for managing flows between UNIX filters and its relationship to the generic flow coordination process of Figure 4-21.

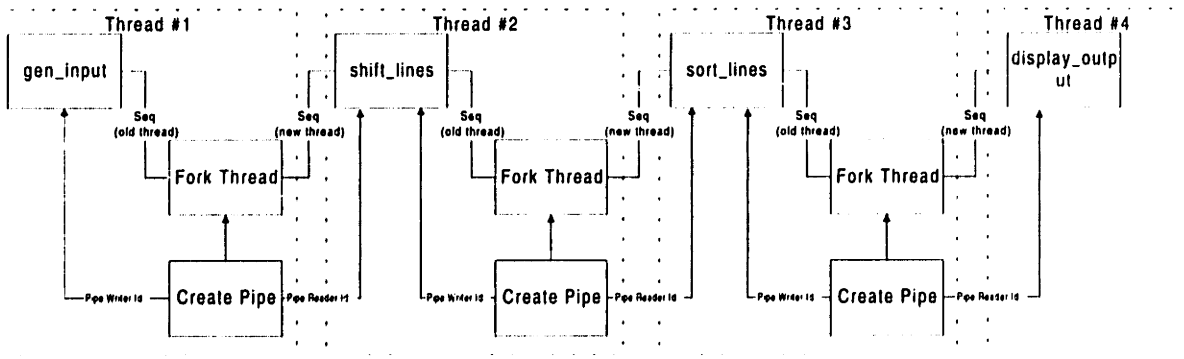


Figure 6-12: Implementation 1 (Filter components interconnected using pipes)

Figure 6-12 shows the resulting organization of Implementation 1.

Implementation 2: Main program-subroutine architecture

Although filter components are designed with pipe architectures in mind, they can also be used in sequential main program-subroutine architectures by using explicit sequentialization to manage the prerequisite between writer and reader. In this case, each filter reads (writes) its entire input (output) stream before control is passed to the next filter. Sequential files, rather than pipes, are used to implement the carrier resource, to avoid deadlock problems with finite capacity pipes. The resulting organization is shown in Figure 6-13.

This combination might be useful for reusing filter components in environments which do not support pipes or concurrent threads.

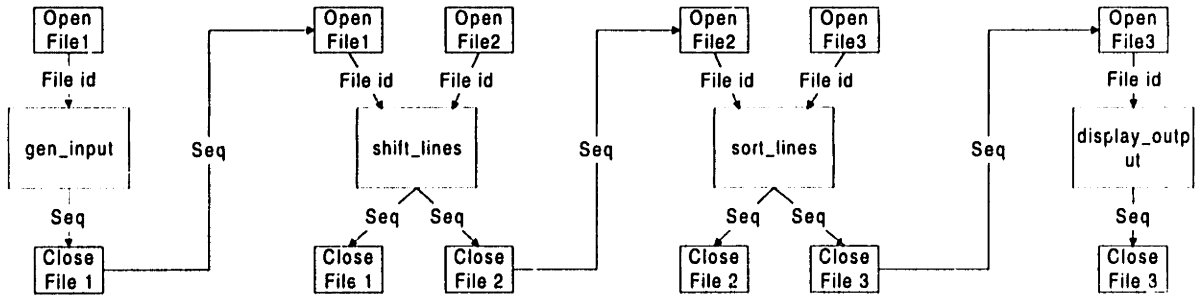


Figure 6-13: Implementation 2 (Filter components organized in a main program-subroutine architecture)

Implementation 3: Implicit invocation architecture

In *implicit invocation* architectures, as defined in [Garlan88], interacting components are being executed by independent threads of control and synchronize using shared events. In the coordination process design space of Chapter 4, this definition corresponds to managing flows and prerequisites using peer coordination processes (see Figures 4-10 and 4-11).

Filters contain built-in support for pipe communication protocols, which have many features in common with implicit invocation protocols. Any other interaction protocol for connecting a filter to other components must be built on top of the pipe protocol: Values written to a byte stream by a writer filter must first be read from the stream, possibly assembled into other data structures, and subsequently passed to the new interaction protocol. At the other end of the protocol, values have to be disassembled into byte sequences and fed into another stream that will transport them to the corresponding reader filter.

Although possible in theory, such an approach achieves nothing except to introduce a redundant additional layer of communication and synchronization. Nevertheless, SYNTHESIS was able to generate a solution even in this case. No sensible designer would select such a solution however, because of its unnecessary complexity and inferior performance relative to the previous two implementations. The resulting organization and detailed code generated by SYNTHESIS in this case are given in Appendix A.2 (Implementation 3).

Set B: Server Components

Implementation 4: Pipe architecture

The generic data flow coordination process of Figure 4-21 can be specialized to handle this combination as follows:

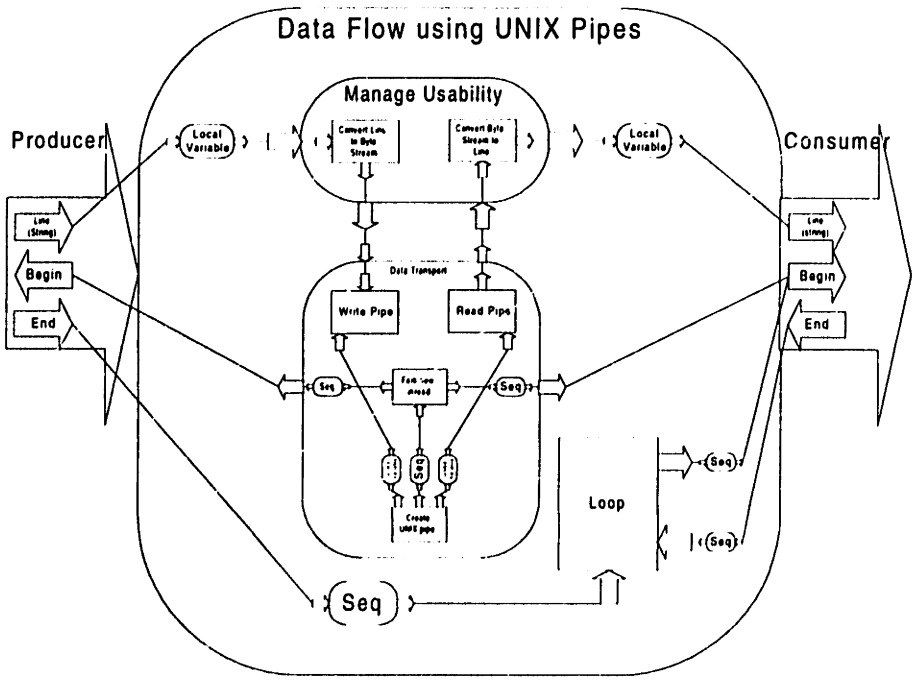


Figure 6-14: Decomposition of a data flow coordination process using UNIX Pipes

- select pipe protocol process to manage Transport Data
- specialize conversion activities of Manage Usability to convert line input to and from byte streams.

The resulting process is shown in Figure 6-14. In this particular experiment, the SYNTHESIS library contained the generic process of Figure 4-21 and a specialization of Transport Data using pipes. It did not contain specializations of conversion activities to transform lines to and from byte streams. During design, SYNTHESIS asked the user to manually add such specializations.

Figure 6-15 shows the resulting organization of Implementation 4.

Implementation 5: Main program-subroutines architecture

The first two flows can be trivially managed by local variables. The interest in this experiment concentrates on the management of Flow 3. In this case the interfaces at both ends of the flow are client interfaces (The sorting component calls `next_sorted_line` in order to output a line of text. The display component also calls `get_next_output_line` in order to retrieve the next line to display). This rules out the possibility of using push or pull organizations to manage the flow (in those organizations, one of the flow participants must play a client role and the other a server role).

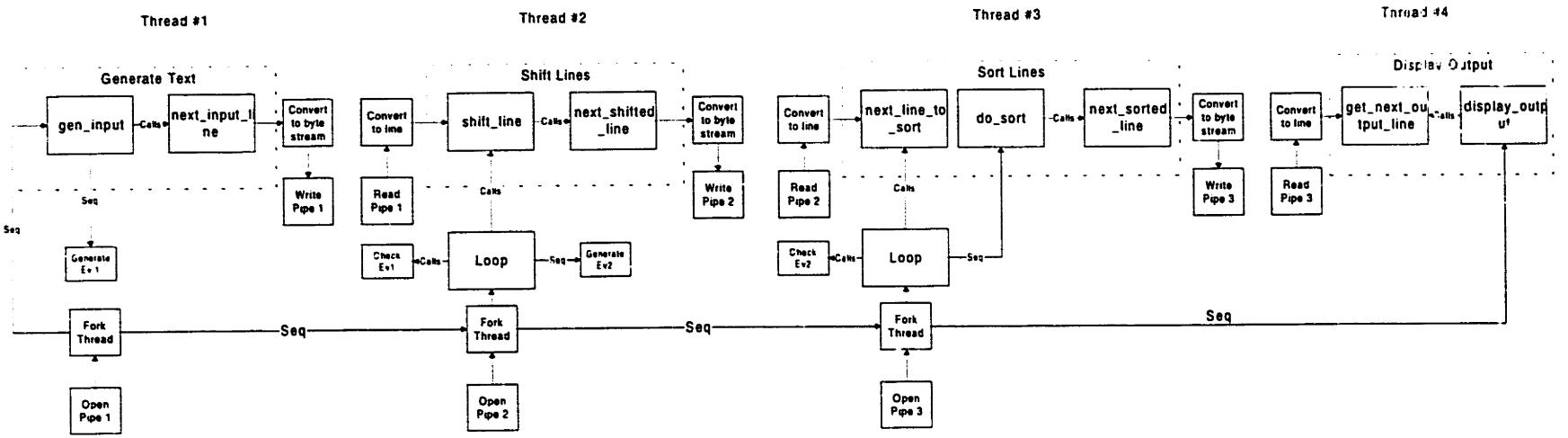
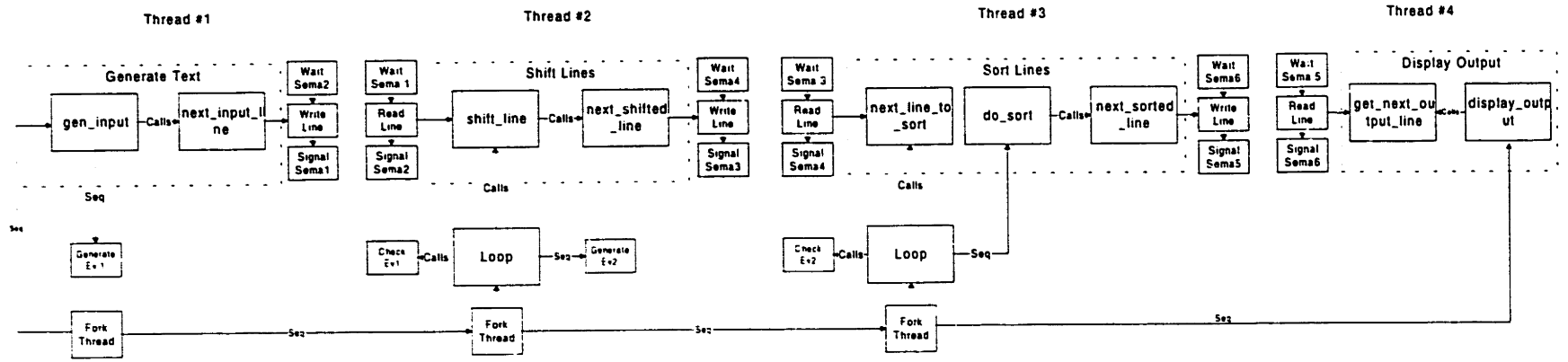


Figure 6-15: Implementation 4 (Server components interconnected using pipes)



* semaphore initialization is omitted for clarity

Figure 6-17: Implementation 6 (Server components in implicit invocation organization)

Peer organizations would require placing the flow participants in different threads, which would violate the main-subroutine organization requirement. The only compatible organization in this implementation is a controlled hierarchy which makes sure that the sorting activity completes before the display activity begins, and uses a shared file for transferring data between the two activities.

Figure 6-16 summarizes the overall organization of this implementation.

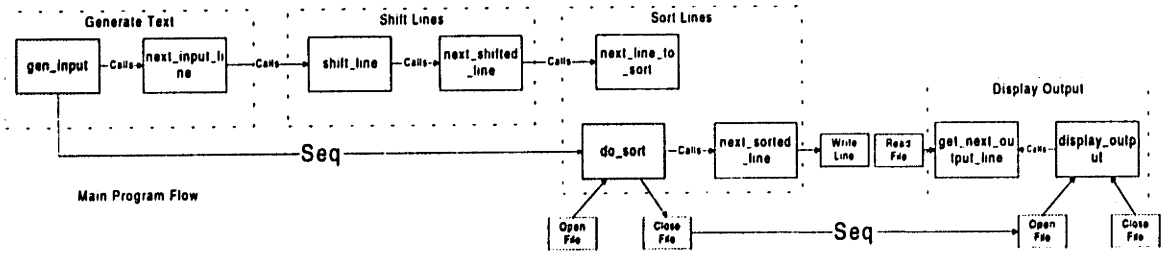


Figure 6-16: Implementation 5 (Server components in main program-subroutine organization)

Implementation 6: Implicit invocation architecture

An implicit invocation architecture can be generated by managing all flows by coordination processes using peer organizations. This requirement translates into managing the prerequisite embedded inside each flow using shared events (see Figure 4-11). In this experiment, we have implemented all flows using a lockstep flow coordination process (specialization of cumulative flow). We have selected a global string variable as our carrier resource and implemented the shared event lockstep prerequisite using semaphores (two semaphores for each prerequisite).

Figure 6-17 shows the resulting organization of Implementation 6.

Set C: Mixed components

Implementation 7: Pipe architectures

We will explain the management of Flow 1. The other two dependencies can be similarly managed. In Flow 1, the producer side is implemented using a filter and the consumer side has a server interface. As explained above this implies that the statement for writing data into the carrier resource is embedded inside the producer. A specialization of a flow with this assumption can be easily constructed from the generic model of Figure 4-21. The specialization which has been selected in this example is shown in Figure 6-18.

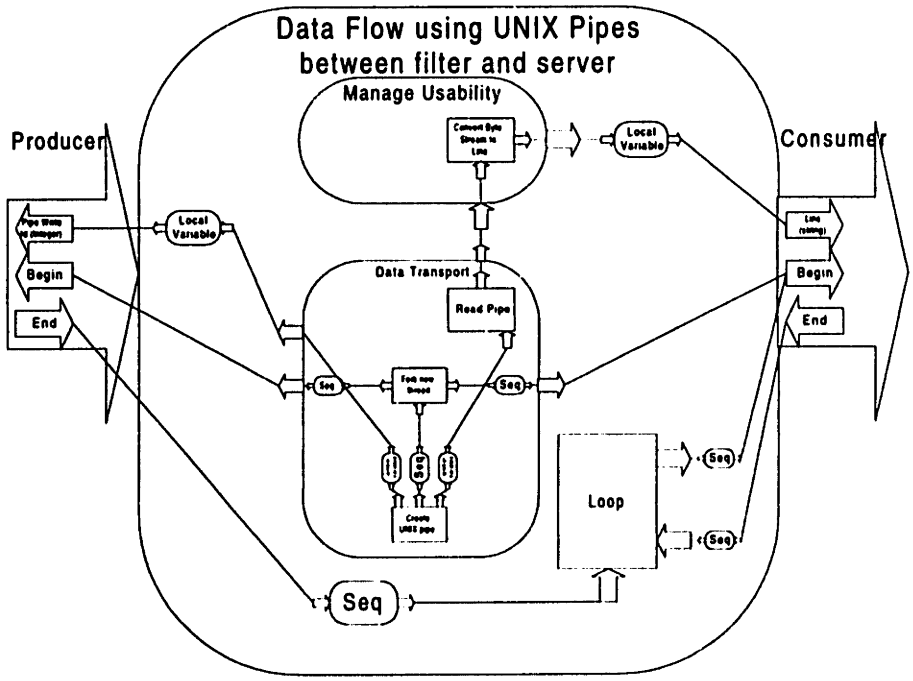


Figure 6-18: Data Flow using UNIX pipes between a filter and a server component.

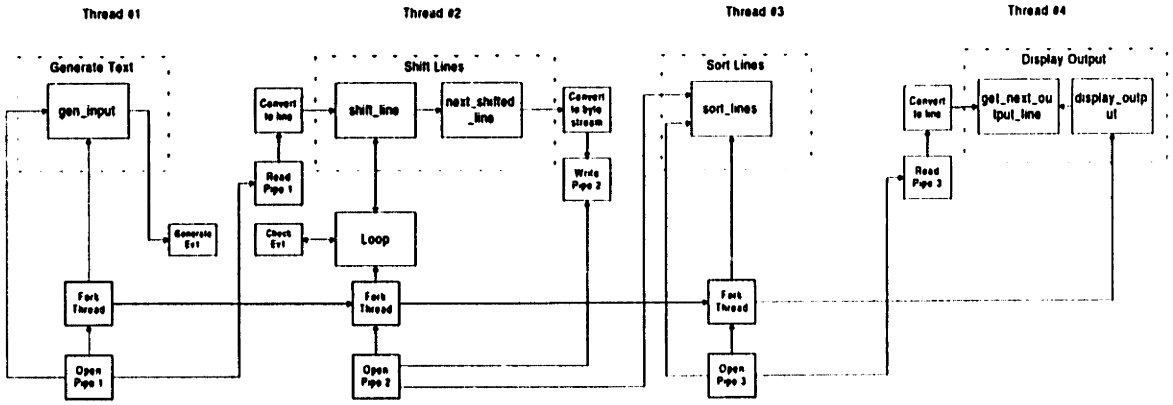


Figure 6-19: Implementation 7 (Mixed components interconnected using pipes)

The overall organization of the resulting implementation is shown in Figure 6-19.

Implementation 8: Main program-subroutine architecture

By using sequential files, rather than pipes, as the carrier resource, filters and server components can be combined into a main program-subroutine architecture using a coordination strategy similar to that of Implementation 2. The resulting organization is shown in Figure 6-20.

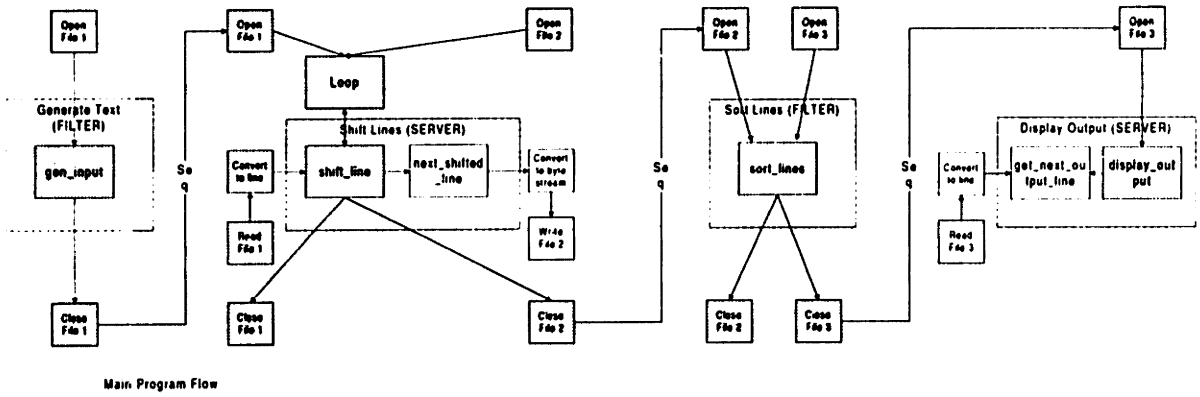


Figure 6-20: Implementation 8 (Mixed components in main program-subroutine organization)

Implementation 9: Implicit invocation architecture

As explained in the description of Implementation 3, filters have embedded support for pipe communication protocols and assume that their flow partners will also be using the same protocol. Any implicit invocation protocol must be built on top of the pipe protocol. This results in an additional, redundant layer of communication and synchronization. For that reason, filter components cannot efficiently be integrated into implicit invocation architectures. Nevertheless, SYNTHESIS was able to generate a solution, even in this case. No sensible designer would select such a solution however, because of its unnecessary complexity and inferior performance relative to the previous two implementations. Interested readers are referred to Appendix A.2 (Implementation 9) for more details.

6.3.3 Discussion

The first experiment (Section 6.2) has shown that SYNTHESIS is able to integrate components with mismatches in expected and provided programming languages, procedure names, and parameter data types, but essentially compatible architectural paradigms and interfaces. This experiment went one step further to demonstrate that the system can resolve more fundamental *architectural mismatches* between components. Architectural mismatches are mismatches in the assumptions each component makes about the structure of the application in which it will be used. The practical consequence of this ability is that it gives designers the freedom to select components independently of their interfaces.

At the same time, the experiment has demonstrated that the overall architecture of an application can be selected to a large degree independently from the architectural assumptions of each individual component. The same set of components can be organized into a set of different architectures, simply by selecting different coordination processes. In this experiment, *the choice of coordination mechanisms, rather than the component implementation, was the defining factor in the resulting software architecture.* One of the

practical benefits of this observation is that it enables designers to use the system in order to port the same set of components to different environments, which might support different interaction and communication mechanisms.

Finally, the experiment has shown that the flexibility of integrating a component in a given overall organization is inversely proportional to the amount of interaction assumptions that are already built-into the component. For example, filter components, which contain more assumptions about their interaction partners than simple procedures, could not be efficiently integrated into implicit invocation (peer) architectures other than pipes. Approaches, such as ours, are very good at adding additional coordination machinery around components. They cannot always *undo* the effects of coordination assumptions already embedded inside components.

In the current implementation of SYNTHESIS, responsibility for selecting the coordination processes which result in the desired overall architecture falls to the designer. One interesting path for future research would be to investigate how different commonly occurring architectural styles can be expressed as constraints on the design dimensions of our coordination process design space. In the preceding description of this experiment, we have informally hinted at some of those constraints (e.g. that implicit invocation architectures constrain the management of flow and prerequisite dependencies by peer organizations only). If we can successfully express architectural styles as coordination design space constraints, our multi-dimensional design space of coordination processes can provide a useful vehicle, both for defining styles as points in our space (combinations of design choices), and for providing more specific guidelines as to which design choices are consistent with a desired architectural style. Finally, our design space could help *invent* and characterize new styles, for which widely-accepted names do not yet exist.

6.4 Experiment 3: An Interactive TEX-based Document Typesetting System

6.4.1 Introduction and Objectives

TEX is a popular document typesetting system, originally developed by Donald Knuth in the late 1970s [Knuth89].The original TEX system was developed for UNIX machines. Today however, there exist versions of TEX for most popular platforms.

TEX was developed at a time when graphical user interfaces and *what-you-see-is-what-you-get* (WYSIWYG) systems did not yet exist. As a result, it was built as a set of independent executable components that are individually invoked from the command line interpreter. TEX components communicate with one another using shared files with standard extensions. The procedure for processing a document using the TEX system can be summarized as follows: Users initially create a text file (with the standard extension .tex), containing the text of the document interspersed with TEX formatting commands. The .tex file is then passed to the TEX processor, which "compiles" all commands and creates a *device independent image file* of the typeset document (with standard extension .dvi). Dvi files can then be previewed by a graphical viewer. Finally, dvi files can be converted to postscript (filename extension .ps) and sent to a postscript printer. Figure 6-21 summarizes the typical invocation sequence of the most important TEX components[1].

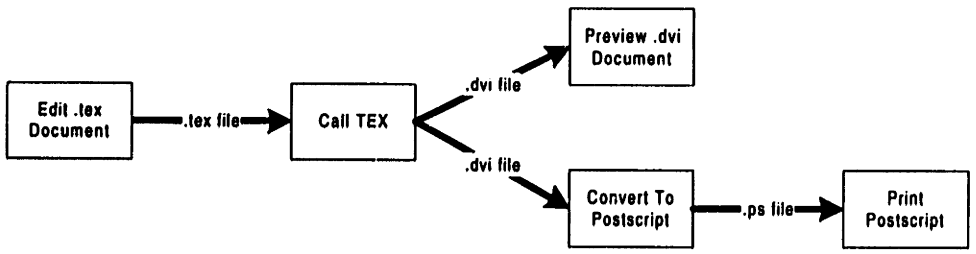


Figure 6-21: Typical sequence of activities in a TEX environment.

In this experiment, we used SYNTHESIS to combine TEX components in a way not intended by their original designers. Instead of sequentially editing the .tex document, invoking TEX, and subsequently calling the viewer, our aim was to build a system that approximates the WYSIWYG behavior of modern word processors: While users are editing the .tex file, the system runs TEX in the background and automatically updates the

[1] For simplicity we have omitted interactions with more advanced components, such as BIBTEX.

dvi viewer. In that manner, the effect of user modifications on the final typeset form of a document should be quasi-instantaneously visible.

Our objectives in selecting this experiment were twofold:

- Test the expressive power of SYNOPSIS and the adequacy of the proposed vocabulary of dependencies for expressing non-trivial patterns of component interaction.
- Test the capabilities and limits of our approach when combining coarse-grained components, such as executable programs, in non-standard ways.

6.4.2 Description of the Experiment

In this experiment, we followed a three-step process:

- Define a generic SYNOPSIS architecture for the application.
- Specialize the architecture to fit the components at hand.
- Use SYNTHESIS to generate coordination code and package all components into an executable application.

The following is a description of the most interesting aspects of each step.

a. Define a generic SYNOPSIS architecture

The first-level decomposition of the SYNOPSIS diagram for an Interactive TEX system is shown in Figure 6-22. Dependencies have been labeled with numbers for easy reference in the text that follows.

The Controller activity represents a graphical user interface that offers three functions to users:

- Open a new .tex document for processing (user supplies filename)
- Print the typeset version of the currently open document (user supplies printer name)
- End the application

Activity Edit Document is responsible for editing the text of the .tex document. It consumes each new .tex filename produced by the Controller, and generates a File Changed control signal whenever the .tex document is modified. It also provides a port for detecting end of application signals.

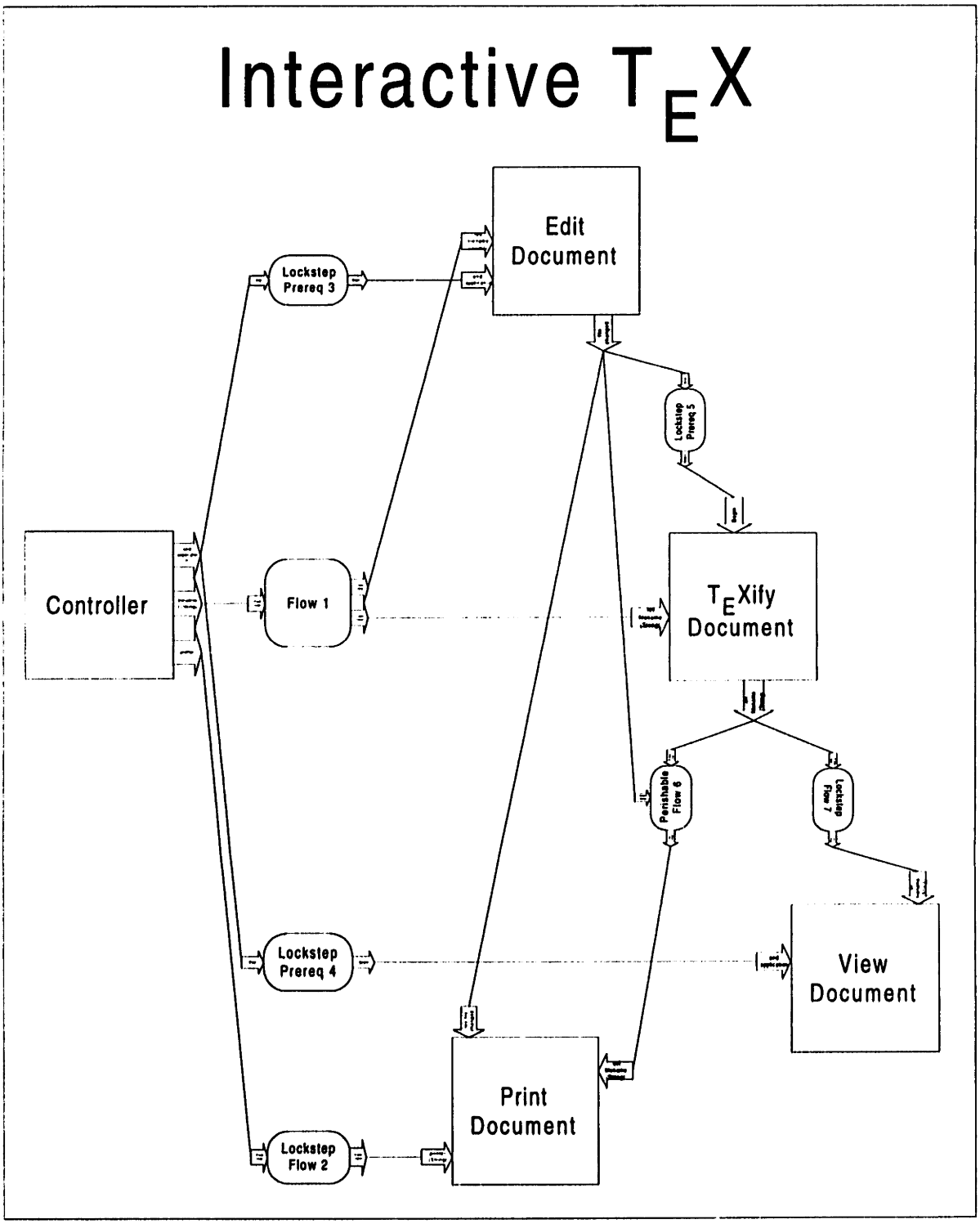
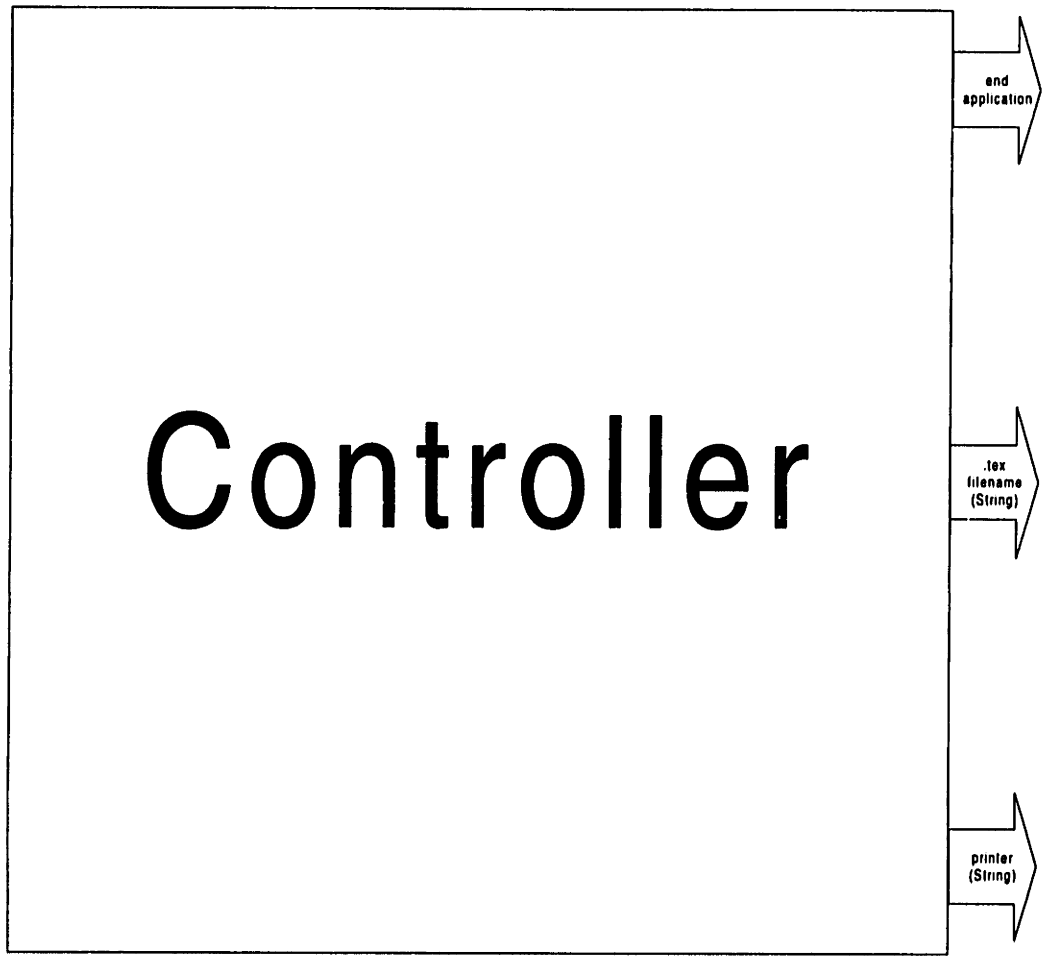
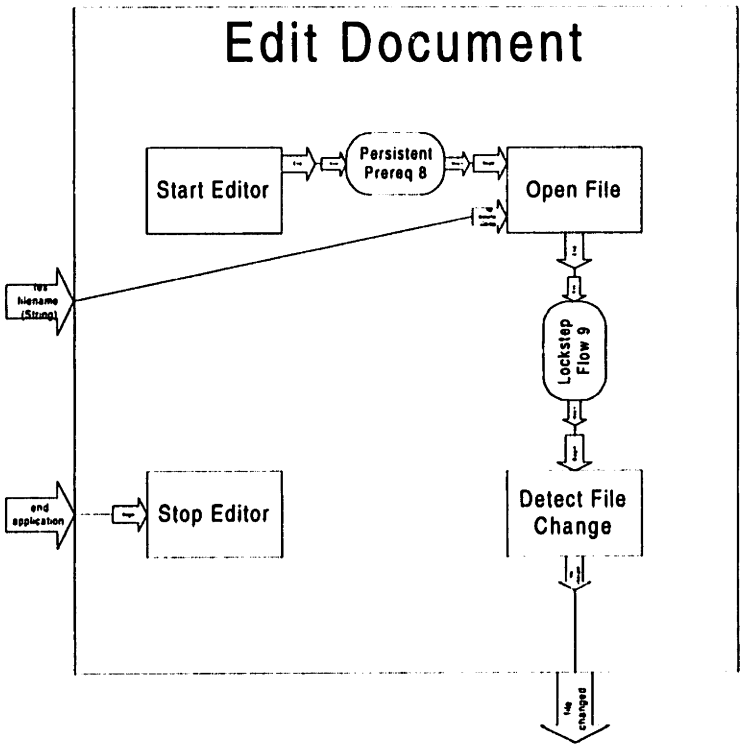


Figure 6-22: First level decomposition of a SYNTHESIS description of an Interactive TEX system.




```
Component "Controller" IsA Procedure

Provides:
  proc tex_main_control();

Expects:
  proc new_document(in filename:DosFileName);
  proc print_document(in printer:String);
  proc end_application();

Files:
  \latex\ctrl.bas
  \latex\latexmen.frm

Attributes:
  Language = vb
```

Figure 6-23: Detailed definition of Controller (user interface) activity.



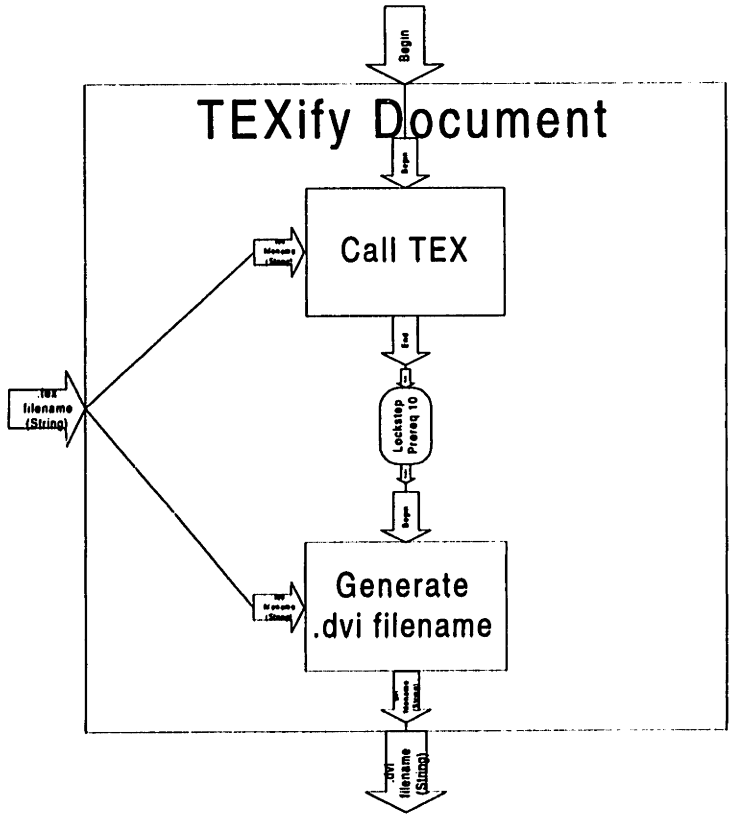

```
Component "Start Editor" IsA Executable

Provides:
  exec msword();

Attributes:
  ExePath = "\applic\msoffice\winword\winword.exe"
```

```
Component "Open File" IsA Gui-Function

Provides:
  gui open_file(in filename:DosFileName);

Attributes:
  guiWindow = "Microsoft Word"
  guiKeys = "^O@l~"
```

```
Component "Stop Editor" IsA Gui-Function

Provides:
  gui quit_word();

Attributes:
  guiWindow = "Microsoft Word"
  guiKeys = "%{FX}"
```

```
Component "Detect File Change" IsA Procedure

Provides:
  proc detect_change();

Expects:
  proc file_changed();

Files:
  \latex\detect.bas
  \latex\detect.frm

Attributes:
  Language = vb
```

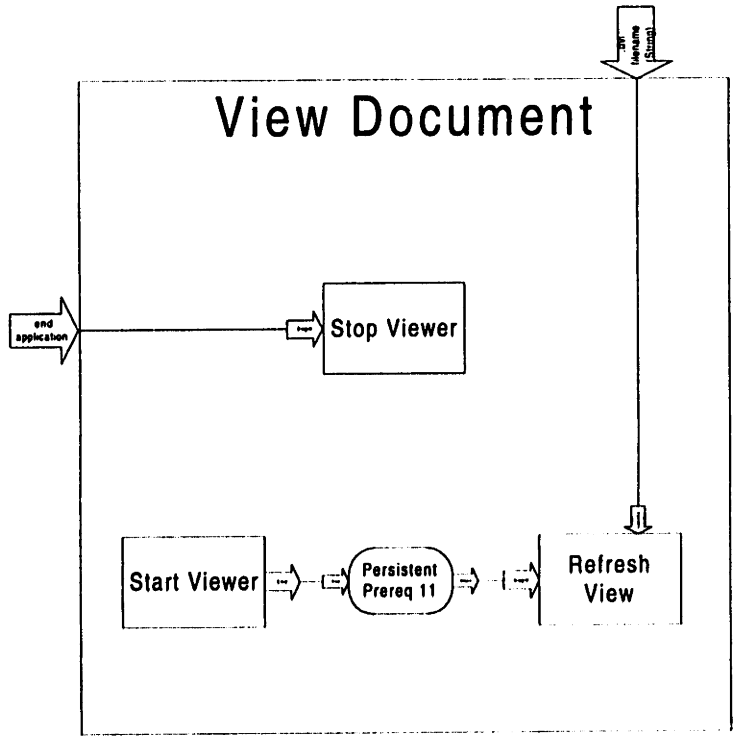
Figure 6-24: Decomposition and component definitions of activity Edit Document.




```
Component "Call TEX" IsA Executable

Provides:
  exec latex(in filename:UnixFileName);

Attributes:
  ExePath = "\emtex\latex.pif"
```

```
Component "Generate .dvi filename" IsA Function

Provides:
  func convert_to_dvi(in filename:UnixFileName):DosFileName

Files
  \latex\convert.bas

Attributes
  Language = vb
```

Figure 6-25: Decomposition and component definitions of activity TEX Document.

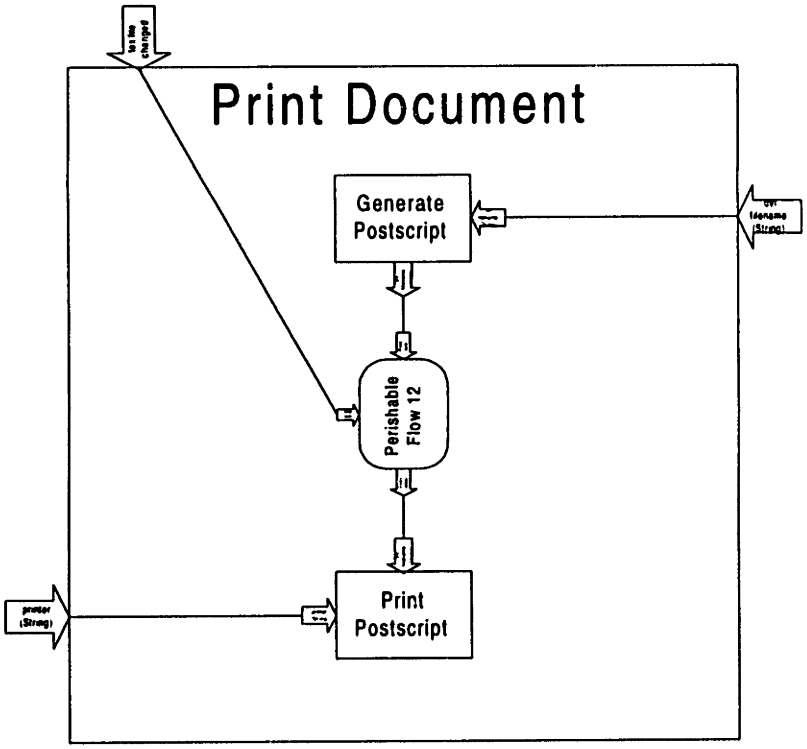


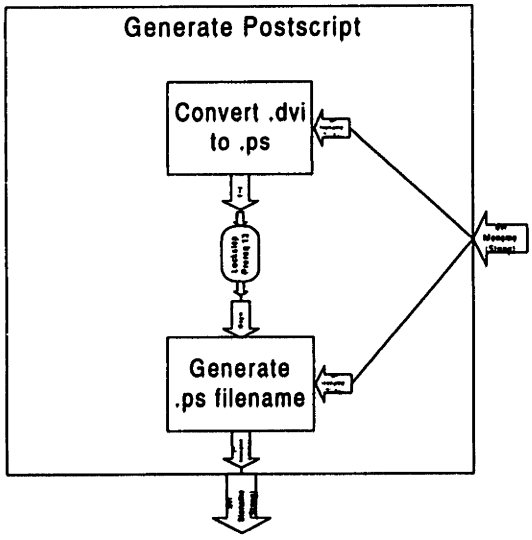

```
Component "Start Viewer" IsA Executable

Provides:
  exec dviwin(attr $Flags);

Attributes:
  Flags = "-1"
  ExePath = "\tex\dviwin\dviwin.exe"
```

```
Component "Start Viewer" IsA Executable

Provides:
  exec dvi_terminate(attr $Termination_Flags);

Attributes:
  Termination_Flags = "-1 -c"
  ExePath = "\tex\dviwin\dviwin.exe"
```

```
Component "Refresh View" IsA Executable

Provides:
  exec dvi_refresh(attr $Flags, in filename:DosFileName);

Attributes:
  Flags = "-1"
  ExePath = "\tex\dviwin\dviwin.exe"
```

Figure 6-26: Decomposition and component definitions of activity View Document.






```
Component "Convert .dvi to .ps" IsA Executable

Provides:
  exec dvips(in filename:DosFileName);

Attributes:
  ExePath = "\emtex\dvips.pif"
```

```
Component "Generate .ps filename" IsA Function

Provides:
  func convert_to_ps(in filename:UnixFileName):DosFileName

Files
  \latex\convert.bas

Attributes
  Language = vb
```

```
Component "Print Postscript" IsA Executable

Provides:
  exec dos_print_command(attr $Cmd,
                         in filename:DosFileName,
                         in printer:String);

Attributes:
  Cmd = "copy"
  ExePath = "c:\windows\dosprmpt.pif"
```

Activity TEXify Document also consumes the .tex filename produced by the Controller. It depends on the File Changed signal produced by Edit Document through Lockstep Prerequisite 5: Every time File Changed is produced (i.e. every time the user modifies the currently open .tex file), TEXify Document begins execution, processes the current .tex file, and produces a new version of the corresponding .dvi file.

Activity View Document is responsible for graphical viewing of .dvi files. Each new version of a .dvi file produced by TEXify Document must flow into View Document and update the graphical viewer. End of application signals terminate the viewer.

Finally, activity Print Document is responsible for printing the currently open document at the user's request. Every time it receives a printer name from the Controller (signaling the issuing of a Print command), it converts the current .dvi file to postscript and sends it to the specified printer.

Since TEX components were intended to be used one at a time, they provide no support for managing sharing or synchronization dependencies. When attempting to integrate them in an interactive system, a number of interesting dependencies arise. Some of them are visible in the top-level decomposition of the diagram:

Activities Edit Document and TEXify Document concurrently use the .tex file, which remains open in the text editor while TEX runs in the background. If one of the components restricts access to the file, special coordination must be installed. This sharing relationship is captured by using a single flow dependency (Flow 1) connecting both file users. As mentioned in Section 4.5.4, the management of sharing dependencies is one of the default decomposition elements of every flow dependency to multiple users.

Modifications in the .tex file invalidate the previous versions of the .dvi and .ps files, which have to be regenerated before they can be used for viewing and printing. This is captured by connecting the File Changed port to the invalidation ports of two perishable flows (Flow 6, visible in Figure 6-22 and Flow 12, visible in Figure 6-27). The role of those flows will become apparent when we discuss the decomposition of Print Document.

b. Specialize the architecture

The top-level architecture of our TEX system is independent of the components that will be used to implement each activity. Lower-level decompositions and mapping of ports to interface elements must take into account the nature of the components that will be used.

We performed this experiment on a Windows machine and selected components available for that environment.

The Controller was implemented using a Visual Basic form showing a set of buttons for each offered function (Figure 6-23). In this experiment, we wrote the (minimal) code implementing the component. However, one can imagine that for such standardized activities, SYNTHESIS could offer a repository of standard components, organized as an activity specialization hierarchy.

Edit Document decomposes as shown in Figure 6-24. For this experiment we selected Microsoft Word as our text editor. MS Word is invoked as an executable, opens a new file when it receives the key sequence: CTRL-O, followed by the filename, followed by newline, and quits when it receives the key sequence: ALT-F, ALT-X.

The most interesting activity is the one that detects file changes. It is a Visual Basic procedure which, periodically queries Microsoft Word (using DDE) to check whether the currently open file is "dirty" (i.e. has been modified since last query). When unsaved changes are detected, the procedure waits for a pause in user activity (detected by a series of consecutive queries that return no new changes). This pause presumably corresponds to the idle time users typically spend between lines or between sentences. When a pause is detected, the procedure directs Word into saving the file and generates the File Changed signal (by calling *expected* procedure `file_changed`).

TEXify Document (Figure 6-25) simply calls the TEX program, passing it the .tex filename, and subsequently generates the corresponding .dvi filename, to be used by the following activities in the chain of events. It is interesting to note that the Generate .dvi filename activity is required to bridge a "mismatch" between the top-level description of TEXify Document and its lower-level decomposition into executable components. At the top-level, TEXify Document is viewed as a black box that consumes a .tex filename and generates a .dvi filename. The actual TEX program, however, does not generate any output filename. In order to "fit" the generic architecture to the components at hand, the conversion activity had to be inserted.

The decomposition of View Document (Figure 6-26) is similar to that of Edit Document. In this experiment we used a publicly available dvi viewer for windows called `dviwin` [Sendoukas94]. `dviwin` provides command line interfaces for starting, refreshing, and terminating itself: Executing the command line:

```
dviwin -1 <filename>
```

starts `dviwin` and makes it display the specified filename. If `dviwin` is already running, the previous command line makes it refresh its display from the current contents of the specified filename. To terminate `dviwin`, users execute the command line:

```
dviwin -1 -c
```

Print Document (Figure 6-27) converts .dvi files to postscript and sends them to a user-specified printer. It is implemented using the DOS command:

```
copy <filename> <devicename>
```

If `filename` is a postscript file and `devicename` is the device name of a connected printer (e.g. `lpt1:`), the above command directs the postscript file to the specified device, without further processing.

Print Document has a number of interesting timing dependencies with the rest of the system. First, a user might request multiple printouts of the same version of a .dvi file. Since conversion of dvi to postscript (performed by a program called `dvips`) takes considerable time, it would not be efficient to repeat it for every print command. For that reason, dependency Perishable Flow 12 was inserted between Generate Postscript and Print Postscript: After a .ps file has been generated for a given .dvi file, it can be used by an arbitrary number of Print Postscript activities. However, every time the .tex file gets modified, the .ps file becomes invalid and must be regenerated from the updated .dvi file. This condition is captured by connecting the File Changed signal coming out of activity Edit Document to the invalidation port of Perishable Flow 12.

There is another, more subtle timing dependency related to the previous one. Whenever a .ps flow becomes invalidated, Generate Postscript must be executed again before further printing can take place. However, Generate Postscript itself reads a .dvi file which has been made invalid by the .tex file modification. Therefore, it also must wait until the latest version of the .dvi file has been generated by TEX. This dependency is captured by the insertion of Perishable Flow 6 between TEXify Document and Print Document at the top-level decomposition of the system (see Figure 6-22). This dependency also ensures that TEX (which writes the .dvi file) and `dvips` (which reads it) do not overlap (Prerequisite dependencies are a specialization of mutual exclusion, see Figure 4-28).

c. Generate an executable application

The next stage in application development using SYNTHESIS involves managing the specified dependencies. Most of the dependencies in this example were easily managed by the coordination processes we have presented in the previous chapters. The following paragraphs discuss the most interesting cases.

The most interesting dependency in this example is the sharing of the .tex file between its two users (Microsoft Word and TEX). Microsoft Word keeps the file open during the background invocations of TEX and, unfortunately, locks it for write access. This behavior is encoded in the filename consumer port of Open New File activity by setting the attribute `Concurrent_Use = False`. The DOS implementation of TEX used in this experiment (`emtex`) for some reason also opens the .tex file for write access. This creates a sharing conflict which cannot be managed by sequentializing access because Word never releases the file lock. One approach that can solve the problem in this case is replication: Each user opens a separate private replica of the file. Before opening a

replica, its contents are updated from the most recently modified replica in the set. For two users, only one replica needs to be created (the other user can open the original file).

Another subtle mismatch in managing Flow 1 of the .tex filename arose from the fact that the command line interface of `emtex`, although written for DOS, receives filenames in the UNIX format (with slashes instead of backslashes). This behavior was indicated to the system by defining two subtypes of `String`, `DosFileName` and `UnixFileName`, and using them in the interface descriptions of the components. The mismatch is handled by the management of the usability dependency. When managing this dependency, SYNTHESIS could not find a conversion activity from `DosFileName` to `UnixFileName` and prompted the user to input one.

The selection of alternatives for managing Perishable Flow 12 inside Print Document (see Figure 6-27) is an interesting example of how SYNOPSIS diagrams and the coordination process design space can help designers visualize and reason about alternative implementations. A Perishable flow can be managed in two general ways: using a consumer pull (lazy flow) organization or using a peer organization, based on shared events (see Section 4.5.3.2). In pull organizations, the producer activity does not take place unless explicitly requested by the consumer. In this example, this would mean that `dvips` (a relatively time consuming process for large documents) would not be executed unless the user needed to print the resulting postscript file. In contrast, in peer organizations, producer activities are executed as soon as their inputs become available and then generate some event to signal their completion. Consumers proceed asynchronously and detect that event if and when they require to use the data. In this example, a peer organization would result in an execution of `dvips` following *each* execution of TEX. Clearly, the pull solution is preferable in this system.

Since all source components in this experiment were written in Visual Basic, SYNTHESIS was able to integrate them by generating 65 lines of Visual Basic coordination code, listed in Appendix A.3.

6.4.3 Discussion

Our previous two experiments focused on testing the power of the coordination process library in resolving interoperability and architectural mismatches between components interconnected through relatively simple dependency patterns. In both cases the architectural descriptions of the target applications were very simple.

The primary objective of this experiment was to test the expressive power of SYNOPSIS and the adequacy of the proposed vocabulary of dependencies in specifying non-trivial applications. After all, the coordination process library can only be used to manage what has already been specified by the designer.

Although judgments of this nature are by necessity subjective, we believe that SYNOPSIS was able to represent the activities and dependencies of this application with clarity and completeness. The generic decomposition of flow dependencies (see Section 4.5) used to manage Flow 1 was able to capture both the sharing and the filename format conversion requirements between the two users of the .tex file. The two perishable flows captured in an elegant way a number of timing dependencies between far-away parts of the system. The resulting graphical descriptions are relatively simple and easy to read.

In this experiment, a number of activities, other than the "main" executable components of the system, had to be manually written. These activities include the Controller (user-interface), the filename conversion activities (Generate .dvi filename, Generate .ps filename) and the file change detection activity (Detect File Change). This requirement reflects the fact that the resulting system offers additional application-specific functionality compared to the original collection of components.

Our system can facilitate the writing of additional activities by letting users concentrate on the implementation of the required functionality and taking care of all details related to compatibility of interfaces. Experiments 1 and 2 provided positive evidence to support this point.

The final application was a useful tool for TEX users, but still fell short of providing a real WYSIWYG capability. One problem was the inevitable delay between the time a user modified the .tex file, and the time that modification was reflected in the dvi viewer. This delay was due to the time it takes TEX to process the .tex file. Unfortunately, TEX can only process entire files. Therefore, a single modification in a single page of a large document can take a significant amount of time to propagate to the dvi viewer.

But even if TEX could instantaneously process documents, another problem would remain. Tex files intersperse document text and TEX commands. Since the synchronization between the editor and TEX is based on relatively simple, TEX-independent heuristics (e.g. the heuristic for generating file changed signals is based on detecting a period of busy time followed by a period of idle time), TEX can be invoked to process a document while the user is in the middle of entering a TEX command. The resulting .tex file at the time of processing will contain TEX "syntax" errors, which might have unpredictable results in the final appearance of the document. Although such situations rarely result in fatal errors, they do make the interaction between editing and viewing TEX files less than seamless.

The root of the problem is due to the coarse-grain nature of the TEX system components. In order to build a truly interactive TEX system we would need finer-grained components, such as a "tex-literate" modifications detector that recognizes when the user finishes entering/modifying a tex command, and a tex processor that incrementally

processes only those parts of a document that change[1]. As we mentioned in Chapter 1, the first consideration in reusing software is *locating* the right components. Our system can help bridge a large number of mismatches when the components are *almost* right, by building coordination code around them. However, as we observed in our discussion of Experiment 2 (Section 6.3.3), the stronger the machinery built *into* the component, the less the flexibility of using the component in arbitrary organizations. Our approach is limited in reusing coarser-grained components in applications which require finer-grained interfaces.

[1] One of the commercial implementations of TEX for the Macintosh, *Textures*, provides a more seamless, "interactive" TEX environment. However, this was accomplished by modifying the code of the TEX components.

6.5 Experiment 4: A Collaborative Editor Toolkit

6.5.1 Introduction and Objectives

Collaborative or *group editors* allow several people to jointly edit a shared document in a distributed environment. They support protocols that control which of the collaborating users are allowed to make changes to the edited document at any one time, how changes are observed by other users, and how they are combined and propagated into the final stored copy of the document.

Collaborative editors have a lot of functionality in common with conventional, single-user editors. In fact, collaboration support can be viewed as an extension of the capabilities of a single-user editor. Therefore, collaborative editor systems can be designed by reusing existing single-user editor components.

In their CSCW '90 paper, Knister and Prakash describe *DistEdit*, a collaborative editor toolkit which provides a set of primitives that can be used to add collaboration support to existing, single-user editors [Knister90]. The primitives provided by the toolkit are generic enough to support editors with different user interfaces. The authors have tested their approach by modifying two editors, MicroEmacs and GNU Emacs, both running under UNIX, to make use of DistEdit.

The procedure for creating a collaborative editor using DistEdit requires manual identification of the points in the editor code where DistEdit primitives should be inserted, small modifications to the editor code in order to fit it to the requirements of DistEdit, and recompilation of the modified editor code plus the DistEdit modules to generate the enhanced executable.

Although it is independent of the user interface of any particular editor, DistEdit depends on the underlying machine and operating system architecture. It only runs under UNIX and makes use of ISIS [Birman89], a UNIX-specific communication package.

In this experiment, we used SYNTHESIS to create an "*architectural toolkit*" for building collaborative editors, loosely based on the ideas of Knister and Prakash. Instead of providing a set of primitive calls to be inserted into the code of an existing editor, our "toolkit" provides a collaborative editor *architecture*, expressed as a SYNOPSIS diagram, to which the original editor must be "fitted". SYNTHESIS can then manage dependencies and generate the final executable system. In contrast to DistEdit, the use of a configuration-independent SYNTHESIS diagram allows the system to generate collaborative editors for several different execution environments and communication protocols, starting from a single description (limited only by the power of the coordination process repository).

Our objectives in selecting this experiment were the following:

- Test the capabilities of the system for constructing "architectural toolkits" that extend the functionality of existing systems, when their source code is available.

- Test the capabilities and limitations of our vocabulary of dependencies for expressing many-to-many relationships.

6.5.2 Description of the Experiment

6.5.2.1 Description of the collaboration protocol

Our experiment implements a collaboration protocol loosely based on the one used in DistEdit. The following is a brief description of the protocol:

The protocol is based on the designation of one of the participants in an editing session as *master*. Master participants have complete editing capabilities. The remaining participants are *observers*, with no editing capabilities. Observers see every change and cursor movement made by the master; the observer's cursor is in "lock-step" with the master's cursor.

Observers cannot perform any operations which change the text. If attempted, such operations simply have no effect.

At all times, at most one participant can be the master. All others are observers. When an editing session starts, there is no master. During that time, any participant can take control and become the master by pressing a designated key-sequence (which might be editor-dependent). During the session, a master may relinquish control by pressing another key-sequence. Once there is no master, all participants are once again allowed to take control.

During time periods when there is no master, all participants are allowed to individually edit their buffers. Each local editing activity is then propagated to all participants. The result is a truly egalitarian mode of collaborative editing.

Any number of users can participate in a collaborative editing session. Participants can enter and leave the session at will, simply by starting or quitting their instance of the editor program. When a new participant enters the session, if there is a master, the current contents of the master's buffer are written back to disk, before they are loaded into the new participant's buffer. In this way, it is ensured that the buffer contents of all participants are identical at all times.

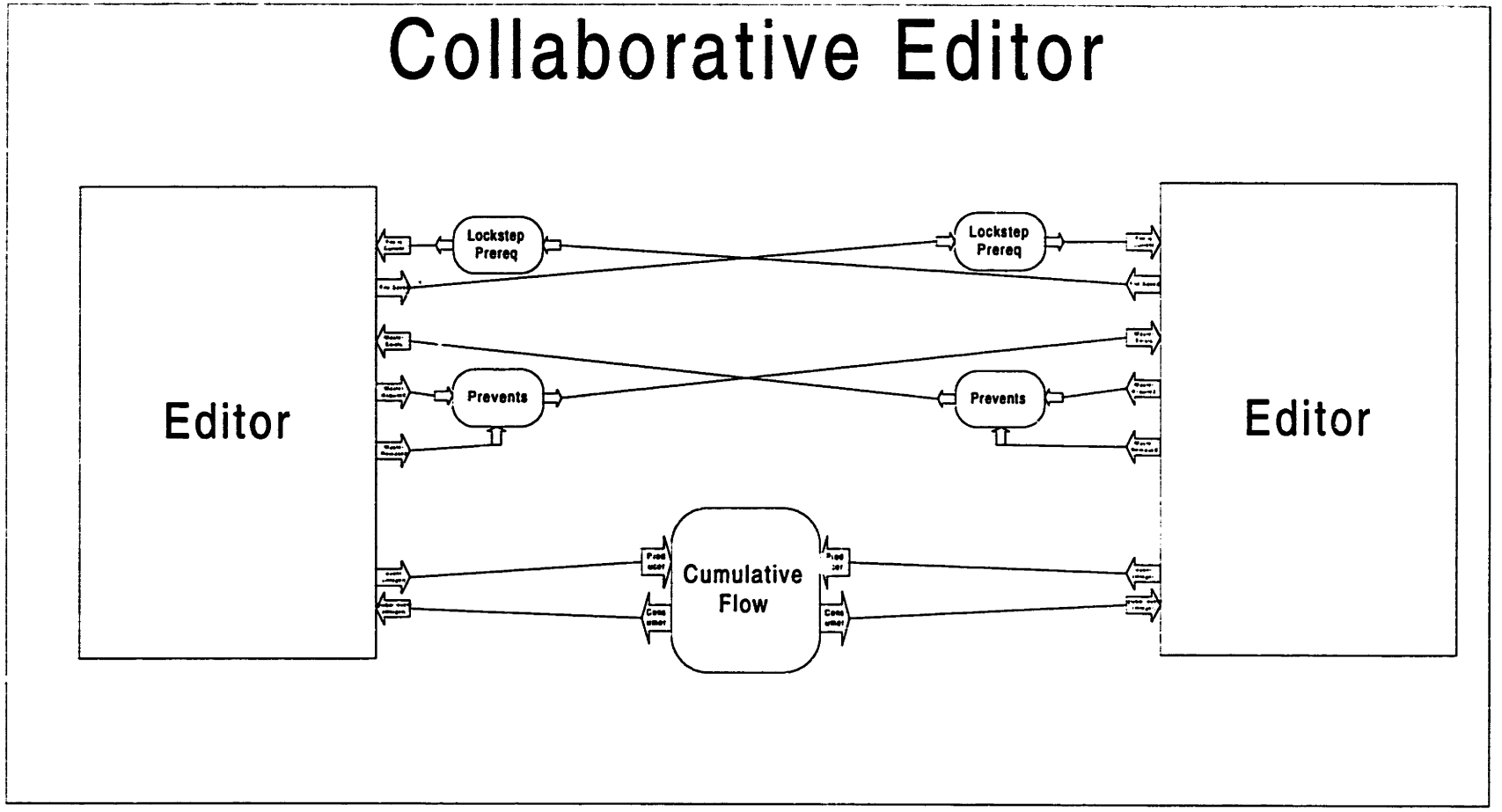


Figure 6-28: Top level decomposition of collaborative editor architecture with two participants.

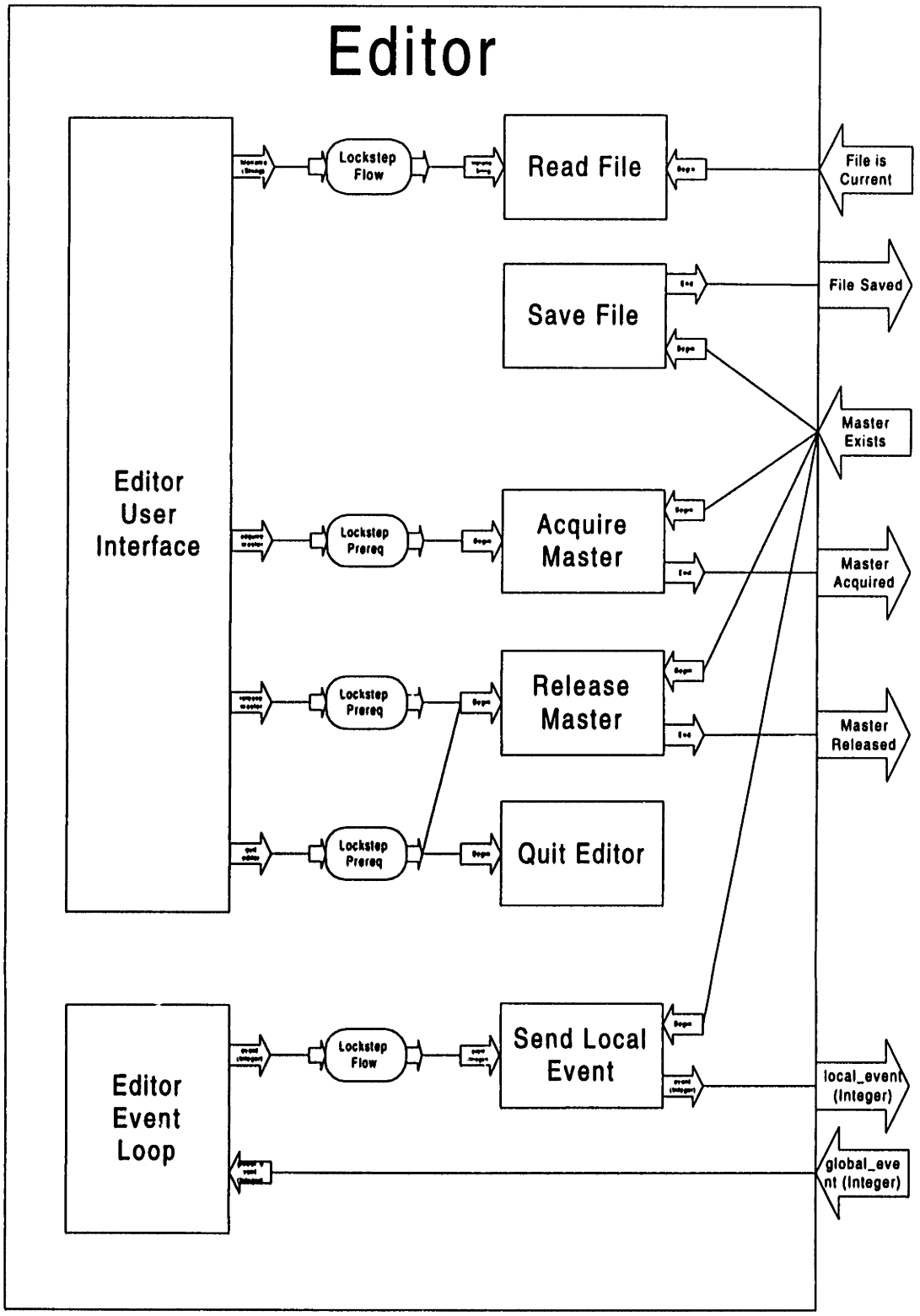


Figure 6-29: Decomposition of Editor activity

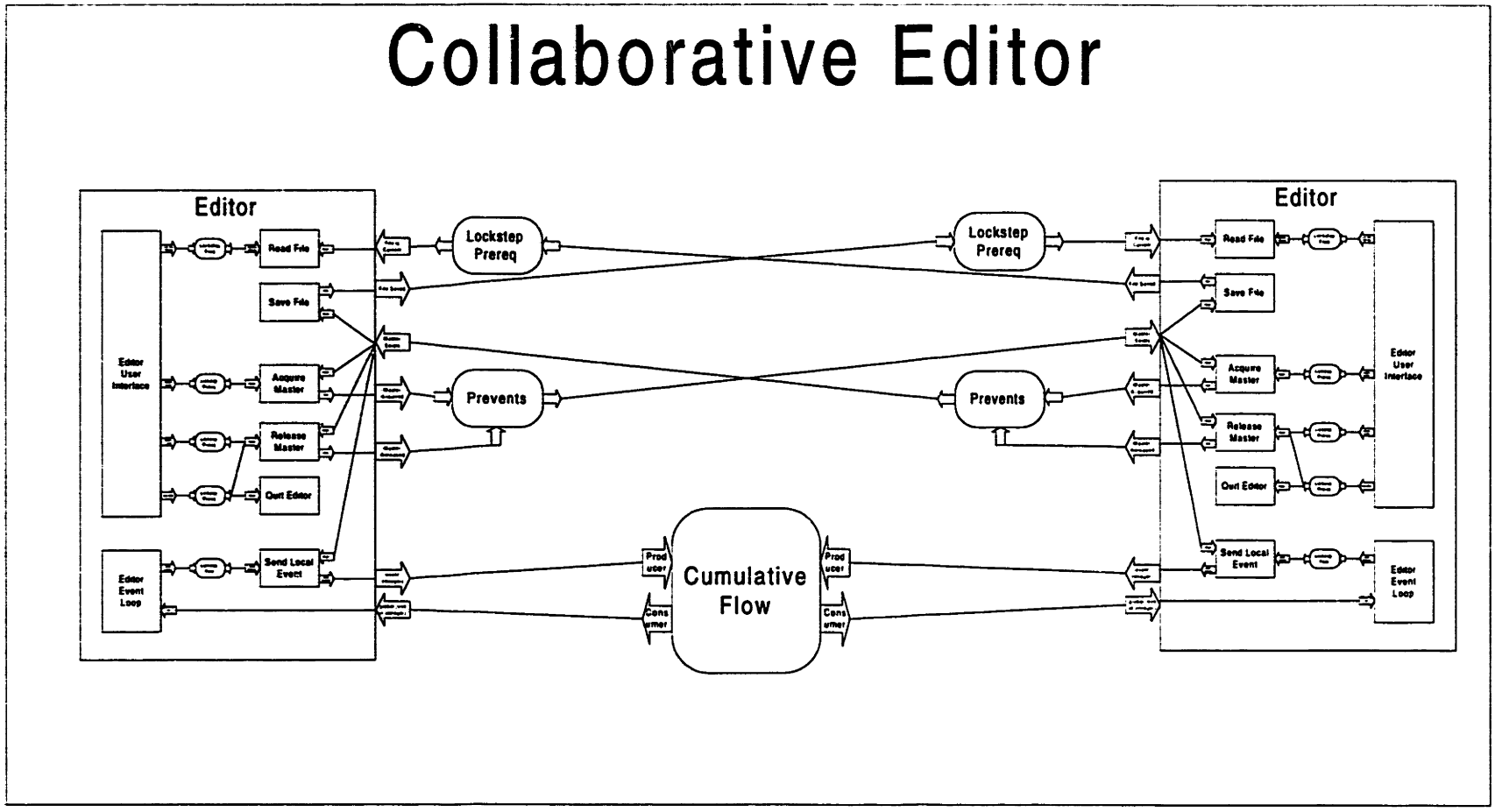


Figure 6-30: Two-level decomposition of collaborative editor architecture with two participants.

```
Component "Editor User Interface" IsA Module

Expects:
  proc read_file(in filename:String);
  proc master_acquire();
  proc master_release();
  proc quit_editor();

Files:
  <listing of all microemacs source files>

Attributes:
  Language = c
  Executable = memacs.exe
```

```
Component "Editor Event Loop" IsA Module

Expects:
  proc local_event(in event:Integer);
  func global_event():Integer;

Files:
  <listing of all microemacs source files>

Attributes:
  Language = c
  Executable = memacs.exe
```

```
Component "Save File" IsA Procedure

Provides:
  proc  filesave(attr $F, attr $N);

Files:
  <listing of all microemacs source files>

Attributes:
  F = 0
  N = 0
  Language = c
  Executable = memacs.exe
```

```
Component "Read File" IsA Procedure

Provides:
  proc  readin(in filename:String, attr $NOLOCK);

Files:
  <listing of all microemacs source files>

Attributes:
  NOLOCK = 0
  Language = c
  Executable = memacs.exe
```

```
Component "Quit Editor" IsA Procedure

Provides:
  proc  quit(attr $F, attr $N);

Files:
  <listing of all microemacs source files>

Attributes:
  F = 0
  N = 0
  Language = c
  Executable = memacs.exe
```

```
Component "Acquire Master" IsA Procedure

Provides:
  proc  m_acq();

Expects:
  proc master_acquired();

Files:
  \memacs\memacs.c

Attributes:
  Language = c
  Executable = memacs.exe
```

```
Component "Release Master" IsA Procedure

Provides:
  proc  m_rel();

Expects:
  proc master_released();

Files:
  \memacs\memacs.c

Attributes:
  Language = c
  Executable = memacs.exe
```

```
Component "Send Local Event" IsA Procedure

Provides:
  proc  m_event(in event:Integer);

Expects:
  proc send_local_event(event:Integer);

Files:
  \memacs\memacs.c

Attributes:
  Language = c
  Executable = memacs.exe
```

Figure 6-31: Component descriptions used to "fit" MicroEmacs into the collaborative editor architecture of Figure 6-29.

Should the master leave the session, it first releases its master status. The remaining participants will then observe a no-master status and be allowed to take control.

When participants leave a session, they can individually select to save or not save the contents of their buffers (which are always in sync). The text being edited can only be lost if all users leave the session without saving it.

6.5.2.2 A SYNOPSIS architecture for collaborative editing

A SYNOPSIS architecture for describing a collaborative editor system which supports the collaboration protocol described in the previous section is given in Figure 6-28. The architecture interconnects a set of simpler Editor activities, corresponding to individual session participants. The decomposition of Editor activities is given in Figure 6-29.

The detailed description of the system can be better understood by referring to Figure 6-30, in which the decomposition of Editor activities has been made visible in the same diagram.

Each Editor activity is based on an existing single-user editor component. The source code of the editor must be available in order to use it in this system. Designers are responsible for "fitting" the source code of the editor to the activity descriptions of Figure 6-29.

The operation of the overall system is based on a set of Prevents dependencies (see Section 4.7), each connecting a participant to all other participants (except itself). Each Prevents dependency is "enabled" whenever a participant acquires master status. It is "disabled" whenever that same participant releases master status. It connects to the Master Exists port of all other participants. While "enabled", a Prevents dependency prevents the execution of all Editor activities connected to the Master Exists port of all participants except the one who acquired master status.

Extending the user interface

Activity Editor User Interface corresponds to the entire source code of an existing single-user editor. In order to "fit" this activity description, the editor source code must be modified in order to call four external (i.e. undefined inside the module) procedure calls that interface it with the rest of the architecture:

- one whenever a new file must be loaded into the editor's buffer
- one whenever the user presses the "acquire master" key sequence
- one whenever the user presses the "release master" key sequence
- one whenever the user presses the "quit editor" key sequence

In most well-designed editors, procedures which handle user command key sequences are collected together in tables and structures that can be easily located by designers. "Fitting" an editor's code to this activity therefore typically only requires changing the names of the "handler" procedures for reading files and quitting the editor, and introducing key sequences and handlers for the two new commands that acquire and release master status.

The following paragraphs describe the effect of each of the previous four commands.

a. Acquiring master status

When a user presses the "acquire master" key sequence, the editor fires the acquire master control port. There are two possibilities:

- If another user has acquired master status, the Prevents dependency connected to the current editor's Master Exists port will prevent the execution of activity Acquire Master. The attempt to acquire master status by the current user will have no effect.

- If no other user is currently holding master status, control passes to activity Acquire Master. This activity does nothing in itself. It simply acts as a switch, which passes control flow on to "enable" dependency Prevents if there is no other master, or does not occur at all, if there already exists a master.

b. Releasing master status

When a user presses the "release master" key sequence, the editor fires the release master control port. The possibilities here are similar to the previous ones:

- If another user has acquired master status, the Prevents dependency connected to the current editor's Master Exists port will prevent the execution of activity Release Master. The attempt to release master status by the current user will have no effect.

- If no other user is currently holding master status, this means that, either the current user holds master status, or no user holds master status. Control then passes to activity Release Master. As was the case with Acquire Master, this activity does nothing in itself. It simply passes control flow on to "disable" dependency Prevents and thus indicate that the current user is no longer the master.

c. Quitting the editor

When a user presses the "quit editor" key sequence, the editor fires the quit editor control port. This passes control to activity Quit Editor. Designers must map this activity to the original source code procedure of the editor for quitting the system.

Typically, this procedure asks users whether to save the current contents of the buffer before quitting.

Before actually quitting, the editor must also release master status, if it is currently holding it. This is achieved by also connecting the relevant prerequisite dependency to activity Release Master, which is executed if no other user is currently holding master status.

d. Reading a file

When a new participant editor starts, or when a user presses the "load file" key sequence, the editor user interface module passes the relevant filename through its filename port. This invokes the Read File activity, which must be mapped to the original source code procedure of the editor for loading a file into its current buffer. However, before this activity can occur, the current contents of the master's buffer must first be saved, to ensure that the new participant loads the latest version of the shared file.

Our architecture expresses this constraint by placing an OR-prerequisite dependency between each Read File activity and every other participant's Save File activity. Save File must be mapped to the original source code procedure of the editor for writing the contents of its current buffer back to disk.

The prerequisite dependency is satisfied if *at least one* of its precedent Save File activities complete. There are two possibilities:

- If there currently exist a master, occurrence of Save File will be prevented at all participants except the master. Hence the prerequisite will be satisfied.

- If there exist no master, all participants will execute Save File. Although this is slightly inefficient, it still guarantees correctness of the protocol.

In order to keep the previous protocol simple, it is executed every time a participant loads *any* new file into its buffers. This might result in unnecessary saves of files in situations where different participants are, for some reason, loading different files. It is assumed that all participants are using the editor to load and edit the shared file only.

Broadcasting master actions

As was mentioned in Section 6.5.2.1, editing activities performed by the master are transmitted to and are immediately visible by all observer participants. In contrast, editing activities attempted by observer participants should have no effect.

At the heart of each editor there exists an *event loop*. An editor basically spends most of its time waiting for user (keyboard or mouse) input. When such input occurs, it is

registered by the event loop as a sequence of events and passed to editor handlers for processing. Events are integer values, corresponding to keyboard codes and mouse activities. Event loops typically wait for events, and place events in an event queue when they occur.

In order to use an editor in a collaborative setting, the function of its event loop must be modified: Local events detected at the master editor must be broadcasted to all participants. Local events detected at observer editors must be discarded. Only global (broadcasted) events received by an editor should be placed its event queue and processed.

Our architecture specifies this required behavior as follows:

Activity Editor Event Loop must be mapped into the modified event loop of each editor. In order to "fit" an editor to our collaborative architecture, designers must modify its event loop in order to

- call a procedure whenever a local event is detected (instead of placing the local event in its event queue)
- call another function for getting the next global event and place each global event returned in its event loop

Once these modifications are in place, the rest of the architecture takes care of actually broadcasting and discarding events.

Whenever a local event is detected by an editor's event loop, there are two possibilities:

- If another user is currently holding master status, activity Send Local Event is prevented from occurring and the local event is discarded. This ensures that observer attempts to edit the shared file have no effect.

- If no other user is currently holding master status, activity Send Local Event is executed. This activity does nothing in itself. It simply acts as a relay which passes its input event on to a many-to-many flow dependency. Management of this dependency will implement the passing of each event detected by the master to all participants (including itself).

Each recipient of a global event returns it back to the modified event loop through the port global event.

In consistency with our initial description of the protocol, when no user is holding master status, the above specification results in the broadcasting of the local events of every participant to all other participants. The use of a cumulative flow dependency means that each editor can proceed at its own pace generating and processing events. The coordination process selected to manage the flow is responsible for storing all pending

events in internal buffers or queues, separate from the local event queues used by each editor instance.

6.5.2.3 Generating executable implementations

We have investigated the capabilities of our architectural toolkit by using it to convert one test system, MicroEmacs [Lawrence89] into a collaborative editor. MicroEmacs is written is C and its source code is available for free. There exist versions of the system for both UNIX and Windows environments. Therefore, it is an ideal candidate for testing the configuration-independence claim of our architectural toolkit.

"Fitting" MicroEmacs to our toolkit required the following manual modifications:

- locating and renaming the handler procedures for loading files and quitting the editor (2 lines of code)

- adding support for acquiring and releasing master status. This involved defining a key sequence and a handler procedure for each new command (2 lines of code)

- locating the handler procedure for saving the current buffer (so that we can map activity Save File to it)

- modifying the editor event loop, as described above. This was the most challenging task because of numerous levels of input abstraction and machine dependencies. A different event loop was used in the UNIX and the Windows version, so each had to be modified separately. The modifications required the writing of 25 lines of code in the UNIX version and 18 lines of code in the Windows version.

The above modifications are very similar to the ones described in [Knister90]. Once the modifications were performed, SYNTHESIS was able generate the additional coordination code needed to generate the final executable applications. The DDE "Wildconnect" feature was used to implement broadcast in the Windows implementation. Sockets were used in the UNIX version. Appendix A.4 lists the generated coordination code.

In this experiment, SYNTHESIS did not perform Stage 3 of the algorithm of Section 5.3, because it is assumed that the editor modules contain all necessary packaging code for starting and initializing themselves. It simply generated the additional coordination code files which were then manually linked with the original editor object files to build the final executable.

6.5.3 Discussion

This experiment has demonstrated the usefulness of SYNOPSIS and SYNTHESIS for building *architectural toolkits* which extend the functionality of existing applications. Architectural toolkits are an attractive alternative to conventional toolkits based on libraries of procedure calls. Their primary advantage lies in their ability to generate alternative implementations, suitable for a variety of different environments and configurations, starting from a single architectural description. In contrast, conventional call-based toolkits are usually restricted to the environments for which they are originally developed.

The process of generating an enhanced application by using a conventional toolkit can be summarized as follows:

- Locate the parts of the application where toolkit calls must be inserted or which must be modified.
- Manually perform the necessary procedure call insertions and modifications.
- Recompile the files of the application plus the files of the toolkit to generate the enhanced system.

The process of generating an enhanced application by using an architectural toolkit is analogous:

- "Fit" the application into the various activities of the toolkit that describe parts of it. This might involve locating specific parts of the application and/or making some modifications.
- Use SYNTHESIS to generate the necessary coordination code for the target configuration.
- Recompile the files of the application plus the files of the toolkit to generate the enhanced system.

As we can see, the manual effort required in both cases is comparable. The advantage of the architectural toolkit approach lies in the fact that it does not require rewriting the toolkit for different environments. It can also be used to generate many alternative implementations for the same environment, to allow experimentation with alternative ways of organizing synchronization or interprocess communication.

From the point of view of the expressing power of our vocabulary of dependencies, this experiment, again, has demonstrated how a non-trivial synchronization and communication protocol can be specified and implemented by the use of relatively simple dependency patterns.

On the other hand, this experiment has indicated a shortcoming of the current SYNOPSIS implementation in expressing many-to-many relationships. Currently, many-to-many dependencies must have a statically fixed number of members. In this example this means we can describe collaborative editors with a fixed maximum number of participants. Figure 6-28 specifies the architecture of an editor with two participants. The diagrams can be easily modified to add more participants (by adding additional instances of the Editor activity and replacing all dependencies with equivalent versions for the desired number of participants). However, the maximum number of participants specified by each diagram must be fixed in advance.

Future research ought to extend the language with constructs for expressing many-to-many relationships between an undefined number of activities.

6.6 Summary and Conclusions

The purpose of this chapter was to demonstrate the feasibility of our approach, and explore different aspects of its usefulness. More specifically, our objectives were to demonstrate:

- the feasibility of describing applications as collections of orthogonal subcomponents, representing core activities and interdependencies

- the usefulness of such a representation, in conjunction with a design handbook of software dependencies, in facilitating both code-level and architectural software reuse.

- the insight provided by such a representation and handbook in exploring alternative ways of integrating a given set of software components

The prototype implementation of SYNTHESIS has proven that the ideas presented in the previous chapters of the thesis can indeed form the basis of a system for developing applications from existing components. SYNTHESIS has also enabled us to carry out the experiments to which the main body of this chapter is devoted.

Experiment 1 (File Viewer) has demonstrated that the system is able to resolve low-level problems of interoperability, such as incompatibilities in programming languages, data types, procedure names, and control flow paradigms. It has also shown how the system can facilitate the exploratory design of alternative component organizations.

Experiment 2 (Key Word In Context) has provided positive evidence for the ability of the system to resolve more fundamental architectural mismatches, that is, different assumptions about the structure of the application in which they will be used. It has also demonstrated that the overall architecture of an application can be specified to a large extent independently of the implementation of any individual component, by appropriate selection of coordination processes. Finally, it has shown that the flexibility of using a component in alternative organizations is inversely proportional to the coordination machinery built into the component.

Experiment 3 (Interactive TEX System) has tested the power of SYNOPSIS and our proposed vocabulary of dependencies in expressing non-trivial application architectures. It has also investigated the strengths and weaknesses of our approach for combining coarse-grained components, such as executable programs into new applications that require finer-grained interaction.

Finally, Experiment 4 (Collaborative Editor) has investigated the usefulness of the system for developing architectural toolkits that extend the functionality of existing source code programs. Architectural toolkits are an attractive alternative to conventional toolkits based on libraries of procedure calls. Their primary advantage lies in their ability to reuse a single architectural description in order to generate alternative implementations, suitable for a variety of different environments.

Overall, the experiments presented in this chapter have demonstrated that that the proposed architectural ontology and vocabulary of dependencies were capable of accurately and completely expressing the architecture of the test applications. Furthermore, they have provided positive evidence for the principal practical claims of our approach, which can be summarized as follows:

- *Support for code-level software reuse:* SYNTHESIS was able to resolve a wide range of interoperability and architectural mismatches and successfully integrate independently developed components into all four test applications, with minimal or no need for user-written coordination software.

- *Support for reuse of software architectures:* SYNTHESIS was able to reuse a configuration-independent SYNOPSIS description of a collaborative editor and the source code of an existing single user editor, in order to generate collaborative editor executables for two different execution environments (UNIX and Windows).

- *Insight into alternative software architectures*: SYNTHESIS was able to suggest a variety of alternative overall architectures for integrating each test set of code-level components into its corresponding application, thus helping designers explore alternative designs.

It is difficult to provide a rigorous proof of the merits and capabilities of a new design methodology for building complex systems. There is certainly room for improvement in every aspect of the ideas presented in this thesis. Nevertheless, the positive results of this chapter suggest that the approach proposed by this research is a step in the right direction, both for facilitating code-level and architectural software reuse, as well as for helping structure the thinking of designers faced with problems of organizing software components into larger systems.

Chapter 7

Related Work

The principal objectives of the work described in this thesis are:

- to support the representation of a software application's high-level functional and interconnection needs, and
- to assist the development of new applications by integrating independently developed software components.

Both are important and difficult problems. Not surprisingly, they have received attention in a variety of different research areas. This chapter describes the various areas which relate to this thesis and explains the similarities and differences between major research efforts in each area and our work. We begin with an overview of the two research areas that have most strongly influenced this work: software architecture and coordination theory. The rest of the chapter describes other approaches for component composition and software reuse and discusses how they relate (or do not relate) to our approach.

7.1 Software Architecture

As the size and complexity of software systems increases, the design and specification of overall system structure, or *software architecture*, emerges as a central concern. Architectural issues include the gross organization of a system, protocols for communication, synchronization, and data access, assignment of functionality to design elements, and selection among design alternatives.

Architectural designs are important for at least two reasons. First, an architectural description makes a complex system intellectually tractable by characterizing it at a high level of abstraction. In particular, the architectural design exposes the top level design decisions and permits a designer to reason about satisfaction of system requirements in terms of assignment of functionality to design elements. Second, architectural design allows designers to exploit recurring patterns of system organization. Such patterns ease the design process by providing routine solutions for certain classes of problems and by supporting reuse of underlying implementations.

While, until recently, the practice of architectural design was largely ad hoc, the topic is receiving increasing attention from researchers and practitioners and is emerging as an explicit discipline within software engineering [Perry92, Garlan94].

Within the emerging field of software architecture research, there are several closely related subareas, including:

- languages for architectural description
- architectural taxonomies and handbooks
- domain-specific software architectures

The following paragraphs will give a brief overview of the above subareas, and will discuss how this thesis relates to each of them.

7.1.1 Languages for Architectural Description

Until recently, system designers had at their disposal two primary ways of defining software architecture: they either used the modularization facilities of existing programming languages and module interconnection languages (see Section 7.3.1), or they described their designs using informal diagrams and idiomatic phrases (such as "client-server organization"). In a number of position papers [Shaw94a, Shaw94b] Shaw has articulated the shortcomings of both those approaches, and the need for specialized architectural description languages.

A number of recent and ongoing research projects are focusing on developing suitable notations for the description of software architectures [Kogut94b]. A common theme in most of these projects is the provision of separate abstractions for describing *components* and *connectors*. Components represent the major computational or data pieces of an application and correspond to implementation-level entities, such as program modules, databases, remote servers etc. Connectors represent mechanisms for describing interactions among components, such as procedure calls, pipes, event systems, blackboard systems, etc.

The two systems that are most closely related to SYNTHESIS are Shaw's UniCon [Shaw95] and Garlan's Aesop [Garlan94b].

UniCon

UniCon provides a general-purpose architectural description language based around two distinct kinds of abstractions: components and connectors.

Components define computational capabilities. They consist of an interface that specifies the capabilities the component exports and an implementation, which may be primitive or composite. Currently supported component types include source modules, shared data, files, filters, and processes.

Connectors mediate interactions among components. A connector consists of a protocol that specifies the class of interactions the connector provides and an implementation. Only primitive connectors are currently supported, although support for composite connectors is planned. Built-in connector types include pipes, sequential file i/o, procedure calls, shared data access, and real time scheduling.

UniCon can express and check appropriate compatibility restrictions and configuration constraints using a mechanism analogous to type checking.

SYNOPSIS has many features in common with UniCon, including a clear separation between activities and coordination processes, the ability to specify and combine multiple component kinds in the same system, and the provision of a mechanism for performing compatibility checks. SYNOPSIS differs from UniCon in two fundamental ways:

- UniCon only provides support for expressing component interconnection patterns (connectors) at the implementation level. SYNOPSIS provides the dependency abstraction which can be used to relate specific interconnection mechanisms (e.g. a semaphore protocol) to the underlying interconnection *needs* that make their presence necessary in the first place (e.g. a prerequisite relationship associated with a resource flow). Dependencies can be defined at various levels of abstraction, from very generic to very specific. The use of generic dependencies enables the construction of architectural descriptions which are independent of a particular machine organization. Such descriptions have two immediate advantages. First, the same generic architectural diagram can be specialized in order to generate alternative implementations for different environments which support different interconnection mechanisms (see Section 6.5). Second, by specifying interconnection needs rather than specific mechanisms, designers can explore alternative ways of organizing their components and select the one that results in better performance (see Sections 6.2 and 6.3).

- SYNOPSIS provides the ability to define new entities (activities, dependencies, ports, resources) as special cases of existing entities using the mechanism of entity specialization (see Section 3.5). Entities defined in this way *inherit* the decomposition and attributes of their parents. They can differentiate themselves from their parents by

modifying their decomposition and attributes, as described in Section 3.5.1. Entity specialization allows the construction of entity hierarchies, which can form repositories of architectural design elements. It also enables the generation of executable systems from generic descriptions of their architecture, by successive specializations of their activities and dependencies.

Aesop

Whereas the focus of UniCon is to make it possible to combine a wide variety of component and connector types within a given system design, Aesop focuses on providing support for developing style-specific architectural development environments. Each of these environments assists and constraints designers into producing applications which conform to a given *architectural style.*

Architectural styles are recurring organizational patterns and idioms, such as pipe-filter, client-server, and event-based architectures. Each style can be expressed by specifying:

- a vocabulary of design elements supported by the style. This is usually a subset of the full vocabulary of components and connectors supported by a general purpose architectural description language. For example, pipe-filter only supports filter component types and pipe connectors.

- a set of configuration constraints, which determine the permitted compositions of those elements. For example, the rules might prohibit cycles in a particular pipe-filter style, or specify that a client-server organization must be a many-to-one relationship.

- optional semantic interpretations, whereby compositions of design elements, suitably constrained by the configuration rules, have well-defined meanings.

Aesop enables the specification of architectural styles using a mechanism similar to subtyping. A style-specific vocabulary of design elements is introduced by providing subtypes of the basic architectural classes or one of their subtypes. Stylistic constraints are then supported by the methods of these types.

Aesop combines the description of a style with a shared toolkit of common facilities to produce an environment, called a *fable*, specialized to that style. Fables are application development environments. They assist and constraint designers into producing applications which conform to the specific architectural style supported by the fable.

SYNTHESIS shares with Aesop the objective of assisting designers to rapidly build correct application architectures. However, instead of assisting designers to conform to a given architectural style, the objective of SYNTHESIS is rather to help them explore and select the architectural style which better suits the requirements of their particular problem. As we have demonstrated in Section 6.3, component implementations might constrain the applicability of particular styles. Furthermore, the optimal organization of a number of

components often requires mixing several different styles in the same system. Restricting designers to using a single style would not enable them to cope with the first situation, or to take full advantage of the second.

Section 6.3 has shown that the overall organization of a set of software components depends to a large extent to the attributes of the coordination processes used to manage each of their interdependencies. As we discussed in Section 6.3.3, the coordination process design space of Chapter 4 might be useful in allowing more precise definitions of architectural styles as set of constraints on coordination process design dimensions. Overall, clarifying the relationship between architectural styles and our design dimensions of coordination processes is a promising path of future research.

7.1.2 Architectural Taxonomies and Handbooks

Most mature engineering disciplines have long ago recognized the importance of classifying and codifying accumulated design knowledge into architectural frameworks and design handbooks. The existence of such handbooks facilitates design reuse and helps focus the design effort to the truly innovative parts of a new system.

Software engineering has only recently recognized the central role of common design patterns and idioms, often referred to by the term *architectural styles* [Garlan94]. Early efforts to identify, name, and analyze these patterns include Shaw's 1989 categorization of a number of idioms [Shaw89]. Garlan and Shaw later extended this list [Garlan94], providing several examples of their use in understanding real systems. Perry and Wolf [Perry92] also recognized the importance of architectural patterns and outlined the use of styles in characterizing applications such as compilers.

A different, but related, area of activity has recently emerged in the object-oriented community through the articulation of (object-oriented) design patterns [Gamma93]. Inspired, in part, by Christopher Alexander's work on pattern languages [Alexander77], these efforts have led to handbooks of common patterns for organizing software [Gamma94, Pree95]. The patterns usually consist of a small number of objects that interact in specific ways. A typical example is the Model-View-Controller pattern, identified by the creators of Smalltalk for organizing user interface software [Krasner88].

Our research focuses on leveraging code reuse by providing a taxonomy and handbook of software interconnection or coordination mechanisms. In contrast to object-oriented design patterns, our taxonomy is independent of any language paradigm and is based on very abstract notions such as resource flow and resource sharing. Language-specific coordination patterns can be expressed as specializations of more generic patterns. For example, a UNIX-specific pipe transfer protocol is defined as a special case of a language-independent protocol for transporting data from a single producer to a single consumer, which, in turn, is defined as one generic process for managing one-to-one data flow dependencies. One of the main objectives of our taxonomy is to allow the

expression of application architectures in a configuration-independent way (e.g. in terms of activities interconnected through resource flow dependencies), that can subsequently be specialized for different sets of components and supported interaction mechanisms.

7.1.3 Domain-Specific Software Architectures

A growing number of industrial research and development efforts are creating *domain-specific architectural styles*, or *reference architectures*, for specific product families [Mettala92]. This work is based on the idea that a common architecture of a collection of related systems can be extracted so that each new system can be built by "instantiating" the shared architecture. Examples include the standard decomposition of a compiler, standardized communication protocols, fourth generation languages (which exploit the common patterns of business information processing), user interface toolkits and frameworks, and various product architectures in domains such as command and control, avionics, manufacturing, and mobile robotics.

SYNTHESIS provides several features that support the development, storage, and reuse of domain-specific software architectures. First, SYNOPSIS architectural diagrams can be specified to be as generic or as specific as desired. Second, they can be stored in the entity repository and reused in other architectures. Third, they can be specialized and refined, in order to incorporate specific components, or to generate coordination code for specific environments. Sections 6.4 and 6.5 describe two examples of domain-specific architectures: one for a TEX-based document typesetting application and one for a collaborative editor. In both experiments, a single abstract top-level architecture was specialized to "fit" the components at hand, and dependencies were managed by appropriate coordination processes for each target configuration.

7.2 Coordination Theory

Coordination theory [Malone94] is an emerging research area that focuses on the interdisciplinary study of coordination. Research in this area uses and extends ideas about coordination from disciplines such as computer science, organization theory, operations research, economics, linguistics, and psychology. Its intended applications include, but are not limited to, understanding the effects of information technology on organizations and markets, assisting the development of successful cooperative-work tools, and helping design efficient distributed and parallel computer systems. In all those domains, a coordination perspective can help (1) analyze alternative designs and (2) suggest new design ideas. Table 7-1 summarizes some sample applications of a coordination perspective.

Coordination theory defines coordination as the process of managing dependencies among activities. Its research agenda includes characterizing different kinds of

Application Area	*Examples of analyzing alternative designs*	*Examples of generating new design ideas*
Organizational structures and information technology	Analyzing the effects of decreasing coordinatio sts on firm size, centralization, a d internal structure	Creating temporary "intellectual marketplaces" to solve specific problems
Cooperative work tools	Analyzing how the payoffs to individual users of a system depend on the number of other users	Designing new tools for task assignment, information routing, and group decision-making
Distributed and parallel computer systems	Analyzing stability properties of load sharing algorithms in computer network	Using competitive bidding mechanisms to allocate processors and memory in computer systems. Using a scientific community metaphor to organize parallel problem-solving

Table 7-1: Sample applications of a coordination perspective (From [Malone94]).

dependencies and identifying the coordination processes that can be used to manage them. Table 7-2 lists a subset of the original list of dependencies identified by Malone and Crowston.

This work is directly related to coordination theory, in that it applies a coordination perspective to software applications. Our work views the process of developing applications as one of specifying architectures in which patterns of dependencies among software activities are eventually managed by coordination processes.

It makes two principal contributions to coordination theory:

- The development of SYNOPSIS, an architectural language that is able to support coordination theory research, with distinct abstractions for activities, dependencies, and coordination processes. Although developed for describing software applications, SYNOPSIS can be used to describe other complex systems as well, such as business processes.

- The definition of a vocabulary of dependency types and a design space of coordination processes for the domain of software systems.

This project grew out of the Process Handbook project [Malone93, Dellarocas94] which applies the ideas of coordination theory to the representation and design of business processes. The goal of the Process Handbook project is to provide a firmer theoretical and empirical foundation for such tasks as enterprise modeling, enterprise integration, and

Dependency	*Examples of coordination processes for managing dependency*
Shared resources	"First come/first serve", priority order, budgets, managerial decision, market-like bidding
Task assignments	(same as for "Shared resources")
Producer / consumer relationships	
Prerequisite constraints	Notification, sequencing, tracking
Inventory	Inventory management (e.g., "Just In Time", "Economic Order Quantity")
Usability	Standardization, ask users, participatory design
Design for manufacturability	Concurrent engineering
Simultaneity constraints	Scheduling, synchronization
Task / subtask	Goal selection, task decomposition

Table 7-2: Examples of common dependencies between activities and alternative coordination processes for managing them. Indentations in the left column indicate more specialized versions of general dependency types (From [Malone94]).

process re-engineering. The project includes (1) collecting examples of how different organizations perform similar processes, and (2) representing these examples in an on-line "Process Handbook" which includes the relative advantages of the alternatives.

The Process Handbook relies on a representation of business processes that distinguishes between activities and dependencies and supports entity specialization. It builds repositories of alternative ways of performing specific business functions, represented at various levels of abstraction. The handbook is intended to function as an *organizational-CAD tool* and helps (a) redesign existing organizational processes, (b) invent new organizational processes that take advantage of information technology, and perhaps (c) automatically generate software to support organizational processes.

SYNOPSIS has borrowed the ideas of separating activities from dependencies and the notion of entity specialization from the Process Handbook. While the Process Handbook project is currently focusing on building a robust and scalable repository of semi-formal descriptions of general business processes, our work focuses on a more narrow domain, software applications, and is especially concerned with (1) refining the process representation so that it can describe applications, coordination processes, and components at a level precise enough for code generation to take place, and (2) populating repositories of dependencies and coordination processes for the specialized domain of software component integration.

7.3 Other Approaches for Component Software Development

7.3.1 Module Interconnection Languages

Module Interconnection Languages (MILs) and *Interface Definition Languages* (IDLs) are an early attempt to provide specialized notations for assisting the structuring of a large collection of modules to form a software system. Representative examples of early language designs include MIL75 [DeRemer75], Intercol [Tichy79] and LIL [Gogouen86]. More recent examples include Stile [Stovsky88] and Conic [Magee89].

MILs provide notations for describing:

- computational units with well-defined interfaces
- compositional mechanisms for gluing the pieces together

A key issue in the design of a MIL is the nature of that glue. In early languages, the predominant form of composition was based on *definition/use bindings*. In this model, each module *defines* or *provides* a set of facilities that are available to other modules, and *uses* or *requires* facilities provided by other modules. The purpose of the glue is to resolve the definition/use relationships by indicating for each use of a facility where its corresponding definition is provided. MILs based on definition/use bindings were intended for use in large but homogeneous systems, where coordination among system components took place using primitive language facilities, such as procedure calls. They focused on the description of components and left the architectural relationships of those components with the rest of the system implicit and often difficult to identify.

In our system, CDL, the notation used in SYNOPSIS to define components and their interfaces (see Section 3.3.1.2), is closely related to the early MILs.

As technology advanced to distributed and heterogeneous systems, simple procedure call bindings were no longer adequate for describing and implementing the range of possible interactions among software components. When attempting to bind modules written in different languages or residing in different systems, a number of new issues (mismatches in data types, communication protocols, etc.) must be addressed by more complex interaction mechanisms. More recent MILs, such as Conic and Stile reflected this situation by providing port-based module interface description languages. Module interaction in those languages no longer relies on procedure call interfaces but rather on a small set of message-passing primitives, which require special run-time support.

Allen [Allen94] and Shaw [Shaw94a, Shaw94b] have made eloquent expositions of the shortcomings of MILs for capturing architectural designs and argued for developing architectural languages with support for complex interconnection patterns. SYNOPSIS is

one such language, providing the abstractions of dependencies and coordination processes, which enable designers to express arbitrarily complex interconnection needs and interaction protocols respectively. The relatively small role that component descriptions play in the total SYNOPSIS description of an application, provides another proof of the limited expressive capabilities of MILs.

7.3.2 Open Software Architectures

Computer hardware has successfully moved away from monolithic, proprietary designs, towards *open architectures* that enable components produced by a variety of vendors to be combined in the same computer system. Open architectures are based on the development of successful bus and interconnection protocol standards. A number of research and commercial projects are currently attempting to create the equivalent of open architectures for software components. Different researchers use different names to characterize their projects, including software bus architectures, object broker architectures, coordination languages, or application frameworks. However, all these approaches are based on standardizing some part of the glue required to compose components. This section gives an overview of some notable efforts in these areas, and contrasts them with the approach taken by our system.

Software bus architectures

Several researchers have attempted to facilitate the interconnection of mixed-language software components by delegating all interfacing and coordination decisions to an abstract decoupling agent, commonly referred to as a *software bus*. Software buses typically offer a "bus interface" to bus clients, implemented as a set of standardized calls that clients use to get specific services from the bus. Common bus "services" include data and control transportation, interface adaptation, and data type conversions. Heterogeneity in languages and architectures is accommodated since program units are prepared to interface directly to the bus and not to other program units. Some notable efforts in this area include Polylith [Purtilo94] and Bart [Beach92].

Object-oriented approaches

Recent object-oriented component software architecture standards, such as CORBA [OMG91], Microsoft's OLE [Microsoft94], and Apple's Open Scripting Architecture [Apple93, Apple94] are all essentially software bus approaches, because they rely on the existence of an intermediary agent or *broker*, through which components interact with one another using standardized interfaces (see [Adler95] for an overview and comparison of the various approaches).

Coordination languages

A variation on the theme of software buses are *coordination languages*, that is, sets of capabilities for interfacing and coordinating software components that can be embedded inside other programming languages [Lucco90, Gelernter92]. The best known coordination language is Linda [Carriero89]. Linda provides a small set of language-independent primitives for thread creation and communication based on a blackboard-like (associative shared memory) data structure called *tuple space*. It relies on special run-time support to implement the tuple space abstraction on top of the physical machine configuration.

Application Frameworks

Application frameworks are guidelines for developing components that can be used together. They typically consist of an *application programming interface* (API), that is, a toolkit of standardized calls that offer intercomponent services, and a set of protocols that guide the way these calls should be used from inside components. The most successful application frameworks are user interface toolkits, such as X-Windows [Scheifler88] and Motif [OSF90], operating system APIs, such as Microsoft Windows API and the Macintosh Toolbox, and rapid application development architectures, such as Microsoft Visual Basic [Microsoft93].

Discussion

Successful software bus approaches can enable independently developed applications to interoperate without the need to write additional coordination code. However, they have a number of drawbacks. First, they can only be used in environments for which versions of the software bus have been developed. For example, OLE can only be used to interconnect components running under Microsoft Windows. Second, they can only be used to interconnect components explicitly written for those architectures.

In contrast, integrating a set of components using SYNTHESIS typically *does* require the generation of additional coordination code, although most of that code is generated semi-automatically. Components in SYNOPSIS architectures need not adhere to any standard and can have arbitrary interfaces. Provided that the right coordination process exists in its repository, SYNTHESIS will be able to interconnect them. Open software architecture protocols can be incorporated into SYNTHESIS repositories as special cases of coordination processes.

7.3.3 Software Schemas

The previous two sections described approaches for facilitating code-level component reuse. However, several researchers are also exploring reuse at a more abstract, architectural level. Their efforts focus on reusing abstract algorithms and data structures

rather than reusing source code. They center around repositories of abstract notations, often called *software schemas*, that represent solutions to commonly occurring software problems, and can be instantiated or specialized to produce source code.

In the software schema approach, the algorithms and data structures captured by the schemas are reusable artifacts. The *abstraction specification* for the schema is a formal exposition of the algorithm or data structure, whereas the *abstraction realization* corresponds to the source code produced when the schema is instantiated. The *fixed part* of the abstraction specification formally describes the invariant computation or data structure of the schema (for example, sorting with respect to an unspecified partial ordering or a queue of an unspecified element type). The *variable part* of the abstraction specification describes the range of options over which a schema can be instantiated, such as the element type and overflow length of a queue.

Examples of software schema approaches include PARIS [Katz89] and the Programmer's Apprentice [Rich90]. The Programmer's Apprentice provides a notation called the Plan Calculus for describing software schemas. A flowchart representation is used to describe algorithmic aspects of a schema, such as control flow and data flow. Flowcharts can be annotated with logical preconditions and postconditions to capture the declarative aspects of the schemas. Empty boxes within a flowchart represent the variable part of the schema abstraction. Software developers instantiate such a schema by filling in the empty boxes with nested schemas.

SYNTHESIS has a number of features in common with the Programmer's Apprentice system. Both systems support a repository of design entities (activities, dependencies, and coordination processes in SYNTHESIS and plans in PA). Just like plans, SYNTHESIS design entities might be executable or generic. In that sense, SYNTHESIS supports both code-level and design-level reuse. Entities decompose into patterns of simpler entities, which can be successively specialized, in order to move from a generic design to an executable implementation.

On the other hand, the two systems support different levels of software design. The Programmer's Apprentice was developed in order to support algorithm and data structure selection and design. Its Plan Calculus notation provides abstractions for supporting conditionals, iterations, and recursion. It does not provide support for complex interconnection protocols and assumes that schema interconnection will take place using primitive language mechanisms, such as procedure calls. SYNTHESIS, in contrast, is designed to facilitate the interconnection of black-box-like components into larger application. It does not yet provide very strong support for expressing elaborate control and iteration structures. It focuses on providing rich abstractions and protocols for component interconnection.

7.3.4 Reusability through Program Transformation

Heterogeneous systems consist of software components written in different languages, using different architectural conventions. Some researchers have pursued transformational approaches for making them work together. That is, instead of writing additional glue code that bridges mismatches between components, they have explored techniques for transforming all components to a common language, or a common architecture.

The TAMPR program transformation system [Boyle84] is an example of this approach. TAMPR is able to translate programs from LISP to Fortran, for reusability in Fortran environments, but also, for improved performance.

Today's operating systems offer rich support for communication and interaction between applications written in different languages. Therefore, program transformation is no longer necessary in order to combine components written in different languages. Section 6.2 has demonstrated how SYNTHESIS can insert appropriate coordination processes around components that, not only resolve differences in programming languages, but also differences in provided and expected procedure names and parameter data types.

On the other hand, Sections 6.3 and 6.4 have demonstrated that the addition of coordination code around components has certain limits when the components have strong built-in interaction assumptions that are incompatible with their intended use in a new system. For example, a component designed as a filter cannot operate well in an interactive environment, no matter how much coordination code is inserted around it. In those cases, manual transformation of component code has to take place. There is no system yet that attempts to identify and automate the transformation of interaction assumptions embedded inside a component's code (other than those directly encoded in a component's interface), but this is an intriguing area for future research.

7.4 Other Related Areas

7.4.1 Theory of Operating, Concurrent, and Distributed Systems

Operating system research is concerned with developing algorithms for the allocation and protection of system resources and the management of communication and synchronization among system processes. The study of concurrent and distributed systems is concerned with essentially the same problems in the special domains of systems with multiple processors or physically distributed components [Andrews91, Bacon93].

Our work provides a unifying framework for organizing the techniques and algorithms developed in those areas, relates them to dependency patterns, and uses them to populate the design space of coordination processes for software systems.

It also encourages a different approach in designing and interconnecting complex systems: Instead of starting with the available communication and synchronization mechanisms of the underlying operating systems and trying to fit the application to them, designers are encouraged to think about the interconnection needs of their systems (in terms of dependency patterns) independently of how these needs will eventually be managed. The repository of coordination processes then assists designers to find a solution which is compatible, both with the underlying interaction needs, and with the available support mechanisms in the target environment. Sometimes this process can suggest solutions which are more effective than those provided by the operating system (like, for example, packaging components in the same executable program and using built-in language coordination mechanisms).

7.4.2 CASE Tools

The term *Computer-Aided Software Engineering (CASE)* covers a wide spectrum of automated tools that support various phases of software design and development. This section will attempt to position our work in the world of CASE tools, by identifying the phases of software design and development that it is able to assist.

Figure 7-1 shows a simplified version of the classic "waterfall" model of software development [Boehm81], excluding the phases involving software testing, installation and maintenance. This model is not a realistic representation of the actual process of developing software. Most notably, it hides the fact that the designers need to revise their designs and thus iterate through its phases. Nevertheless, it still serves to identify the major phases in the design and development of a software system. These phases are briefly explained in Table 7-3.

Figure 7-1 also shows where SYNTHESIS fits with respect to the waterfall model. As can be seen, it has the potential of assisting all phases of software development, except the initial requirements analysis.

SYNOPSIS can be used to sketch a decomposition of a system into generic activities, which can then be specialized and refined as more insight about the system is gained. SYNTHESIS can store repositories of alternative domain-specific architectures that can further assist designers both in specifying and in refining the functional decomposition of their systems.

Once designers have finalized the decomposition of an application into patterns of activities and dependencies, SYNTHESIS can assist them to select executable

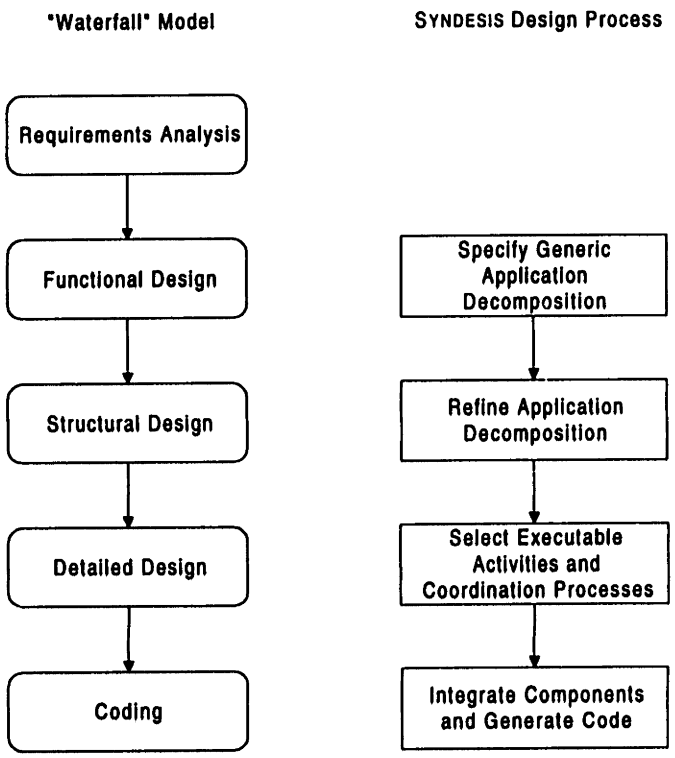


Figure 7-1: Waterfall model of software development and corresponding stages of application development using SYNTHESIS.

Requirements analysis	Identification of the application(s) to be supported and the user community to be served. Important characteristics to be determined may include the sophistication of the users, level of user-machine interaction desired, performance constraints, etc.
Functional design	Identification of the major functions and goals the system has to meet
Structural design	Refinement of system functions and division into modules
Detailed design	Selection of major algorithms and data structures within each module, selection of communication and synchronization protocols between modules
Coding	Implementation of modules. Integration of modules into an executable system

Table 7-3: Phases of software development in the waterfall model

implementations for atomic activities, and coordination processes for managing dependencies through the process we described in Chapter 5.

Finally, when detailed design is complete, SYNTHESIS can automatically generate coordination code, package existing components into procedures and executables, and produce the final executable system.

Chapter 8

Conclusion

This chapter summarizes the results of this work and distills a set of design guidelines for developing notations, mechanisms, and tools that facilitate software reuse. It concludes by discussing some ideas for future research.

8.1 Summary of Results

This thesis began with the question: Why is it difficult to integrate existing components into new applications?

Chapter 2 attempted to provide one answer. Our answer was: Because components built using current technologies contain fragments of interconnection protocols that connected them to other components in their original applications. This is due to a failure of current programming languages to provide separate abstractions for representing interconnection protocols among components. Programmers are thus forced to disperse fragments of such protocols among the interdependent components. When attempting to reuse such components in applications with different interconnection needs, these built-in, and often undocumented, assumptions get in the way. They have to be manually identified and resolved, either by modifying the code of the component, or by manually writing additional coordination code that bridges assumption mismatches.

This observation suggested one solution for alleviating the problem: Separate the "core" function of components from their application-specific interconnection protocols and develop language abstractions for specifying and implementing each separately. The main body of this thesis was devoted to proposing and exploring one practical way of achieving this separation.

In order to be able to separate the main functional pieces of an application from their interconnection needs and protocols, we need notations for describing application architectures that provide separate abstractions for each. Chapter 3 introduced SYNOPSIS, one such notation developed for this thesis. SYNOPSIS is an architectural language that supports two orthogonal abstractions: activities, for describing the main functional pieces of an application, and dependencies, for specifying their interconnection needs in the context of an application. Dependencies are managed by coordination processes, an attribute of dependencies which represents implementations of interconnection protocols. Using SYNOPSIS, designers specify new applications as patterns of activities, interconnected by dependencies. Applications can then be implemented by successively specializing activities and dependencies, until all activities are directly associated with code-level software components, and all dependencies are managed by executable coordination processes.

To assist the design task of representing application interconnection needs, as well as the design of appropriate coordination processes, Chapter 4 proposed a standardized, but extensible, vocabulary of dependency types and an associated design space of coordination processes. The vocabulary is based on the observation that the most frequently occurring patterns of component interdependencies are independent of any application domain, and can be specified using a relatively narrow set of concepts, such as resource flows, resource sharing, and timing dependencies. Likewise, the design of coordination processes involves a relatively narrow set of concepts orthogonal to the problem domain of most applications, such as machine architectures, language conventions, and communication mechanisms. For those reasons, the development of an application-independent framework that captures the most useful patterns of interdependencies and the ways of managing them, looks like a feasible and useful endeavor. Such a framework can form the basis for a developing a design handbook of software component interconnection.

To test the feasibility and usefulness of both our language and our design space, we built a prototype component-based application development tool called SYNTHESIS. SYNTHESIS provides graphical editors for entering and editing SYNOPSIS architectures. It provides support for building repositories of SYNOPSIS entities, including activities, dependencies, and coordination processes. Finally, it provides a design assistant that exploits a repository of dependencies and coordination processes based on the framework of Chapter 4, in order to semi-automate the design process specializing generic design elements, and automate the process of integrating a set of excutable design elements into sets of code modules. Chapter 5 was devoted to a detailed description of the algorithms and transformations used by the design assistant.

In Chapter 6, we have used SYNTHESIS to perform four experiments that tested different aspects of our approach. The most important areas where we have put our technology to the test include:

- the adequacy of SYNOPSIS and our current vocabulary of dependencies in describing non-trivial applications accurately and completely

- the extent to which our technology is capable of facilitating *code-level software reuse*. This was tested by using the system to compose independently developed components into new applications, with minimal need for manual modifications or additional user-written code, and

- the ability of our technology to assist *architectural-level software reuse*: This was tested by generating alternative implementations, suitable for different execution environments, from a single SYNOPSIS description.

SYNOPSIS and our current vocabulary of dependencies were able to concisely and elegantly describe all four test applications. SYNTHESIS successfully exploited its repository of coordination processes in order to generate all four systems with minimal need for additional user-written code. Additional user-written code was only required in three cases to implement unsupported data format conversions, and consisted of simple subroutines which, once written, became a permanent part of the coordination processes library and were re-used in later experiments.

The system was able to resolve both low-level *interoperability mismatches* (differences in provided and expected procedure names, parameter data types, languages, etc.) between heterogeneous components, as well as more fundamental *architectural mismatches* between components with incompatible built-in interaction assumptions (e.g. a filter, writing sequential byte streams, and a server, processing individual lines of text). The stronger the built-in assumptions, however, the less the flexibility of efficiently reusing a component in alternative organizations.

Finally, our experiments proved that describing applications at the level of activities and dependencies allows a single SYNOPSIS description and a single set of components to be reused for generating applications for different target environments simply by selecting different coordination processes suitable for each environment.

8.2 Design Lessons

The main body of the thesis is devoted to describing one concrete approach for developing software applications from existing components. Underlying this approach, however, is a set of more abstract principles that can serve as a framework for developing notations, mechanisms, and tools for facilitating software reuse. This section distills these abstract principles and gives a brief explanation.

- *Make architectural assumptions explicit.* Current programming languages lack notations which clearly state what assumptions each component is making about the application in which it is to appear. Such assumptions might include how and from

where a component expects to receive its inputs, how and where it expects to send its outputs, whether it expects to have exclusive or shared use of certain resources, the amount of infrastructure it expects to find in its target environment, etc. This thesis has explored the mechanism of ports and attributes for expressing component assumptions.

- *Construct large pieces of software using orthogonal subcomponents.* Methodologies are needed that decompose software systems in a way that allows its constituent components to be independently developed, specialized, or replaced. This thesis has explored the merits of one such decomposition, based on separating core functional pieces (activities), from their patterns of interconnection (dependencies).

- *Develop systematic design libraries for component interconnection.* Even with good documentation and decomposition into sets of orthogonal entities, mismatches will inevitably occur. In the past, such mismatches were dealt by manually modifying component code, or by introducing additional coordination software. A more systematic approach is clearly needed. This thesis has proposed a framework for designing coordination processes, which attempts to organize a large number of practical techniques as special cases of a relatively small set of principles. The proposed framework can form the basis of a design handbook of software component interconnection problems.

8.3 Future Work

In this section I will briefly describe some ideas for further research suggested by the various components of this thesis.

8.3.1 SYNOPSIS Architectural Language

No programming language design is ever complete. As more experience is gained with a programming language, additional features are added and existing features are modified to enrich its expressive power. We expect the same to happen with SYNOPSIS.

Two immediate areas of future enhancements have been identified in the thesis:

- *Support for indefinite sets.* As pointed out in Section 6.5, all SYNOPSIS activities and dependencies currently must have a statically fixed number of ports. In other words, each activity must know at design time how many (and which) other activities it might interact with. In many real systems, interaction patterns are determined dynamically during run-time and might vary during the lifetime of an execution. To be able to model such systems, SYNOPSIS must be extended with activity and dependency constructs that represent sets of indefinite (run-time determined) cardinality.

- *More systematic expression of architectural assumptions.* SYNOPSIS is using attribute definitions for expressing architectural assumptions. Although attribute definitions seem to be powerful enough to express a number of relationships and constraints (see Section 3.4), the current system does not provide systematic guidelines on how and when to use them. More research is needed to classify architectural assumptions, and standardize the way these assumptions are expressed in SYNOPSIS. An informal taxonomy of interaction assumptions given in Section 2.3.1 can serve as a starting point.

8.3.2 Coordination Process Design Space

Extend vocabulary of dependencies

Although the vocabulary of dependencies presented in Chapter 4 is capable of describing a large number of commonly occurring relationships, it by no means claims completeness in any rigorous sense. Further experience with using the approach to describe and implement non-trivial software applications might uncover additional relationships that could usefully be encoded as special cases of existing dependency types, or as new dependency types. Likewise, the design space of coordination processes can be enriched by new generic processes, or by new special cases of existing generic processes.

A particularly promising path seems to be the discovery and classification of commonly occurring composite patterns of dependencies, for which efficient joint coordination processes have been developed. One example is the joint management of a set of unidirectional flows through the network by combining the respective data items into a single packet. The existence of a library of such composite entities will enable automated design assistants to scan SYNOPSIS application diagrams, discover patterns of simpler dependencies that correspond to composite dependencies, and efficiently manage them using joint coordination processes. Section 4.8 presents some examples of composite dependencies and coordination processes for jointly managing them.

Develop coordination process design rules

In the current implementation of the system, designers are responsible for selecting among multiple compatible coordination processes for a given dependency. It would be interesting to develop design rules that help automate that selection step by ranking candidate processes according to various evaluation criteria such as their response time, their reliability, and their overall fit with the rest of the application. For example, when managing a data flow dependency, one possible design heuristic would be to use direct transfer of control (e.g. remote procedure calls) when the size of the data that flows is small, and to use a separate carrier resource, such as a file, when the size of the data is large.

Explore relationship to architectural style

Several researchers are using the term architectural style to describe and classify recurring organizational patterns and idioms, such as client-server, pipe-filter, and event-based architectures. Section 6.3 has provided some evidence for the important role of coordination processes in determining the overall architectural style of a software system: By selecting different coordination processes, the same set of components can be organized into different styles.

Our multi-dimensional design space of coordination processes can thus provide a useful vehicle, both for defining styles as points in our space (combinations of design choices), and for providing more specific guidelines as to which design choices are consistent with a desired architectural style. For example, in Section 6.3 we hinted that event-based architectures exclude the management of dependencies using push or pull organizations. Furthermore, our design space could help invent and characterize new styles, for which widely-accepted names do not yet exist.

8.3.3 SYNTHESIS Design Assistant

Like every prototype implementation, SYNTHESIS could benefit from a substantial rewrite that will focus on improving its performance and scalability. From a research perspective, the following are some interesting topics of future work:

Develop search/selection heuristics

Before managing a dependency, the SYNTHESIS design assistant performs an exhaustive search of candidate coordination processes, applying the compatibility checking algorithm of Figure 3-15 to each candidate. For large repositories of coordination processes, this approach will be prohibitively slow. Appropriate heuristics must be developed that will eliminate some of the potential candidates without the need to apply the compatibility checking algorithm. Such heuristics might rely on specifications of performance requirements, or on constraints on the desired architectural style of the overall system.

Build handbook of domain-specific architectures

In this thesis, we have emphasized the use of SYNTHESIS for building repositories of application-independent coordination processes and assisting designers in the management of dependencies. In most of the thesis we have assumed that the responsibility for decomposing the architecture of a system into a pattern of activities and dependencies fell to the designer.

However, both SYNOPSIS and SYNTHESIS are particularly well-suited for building repositories of generic and specialized architectures in a wide variety of domains. Such

repositories of domain-specific architectures or *software architecture handbooks*, could serve as useful starting points for designing any new system: When starting the design of a new application, designers first consult the handbook, in order to retrieve alternative generic architectures of similar systems. They can then specialize those generic architectures to fit their particular needs.

The use of the system as a software architecture handbook is an intriguing path of future work. Additional considerations that will have to be addressed in this case include building a robust and scalable SYNOPSIS entity repository, and providing adequate navigation mechanisms for retrieving related architectures.

8.3.4 New Component Programming Languages

This thesis argued for notations which separate the "core" function of software components from their application-specific interconnection protocols. It proposed SYNOPSIS, an architectural language for describing new applications with these properties. SYNOPSIS allows the main functional pieces of a new application to be specified independently from their patterns of dependencies in that application.

Since the practical objective of our system is to facilitate the reuse of *existing* components in new applications, SYNOPSIS executable activities are currently associated with code-level components built using current technologies. Such components, inevitably incorporate some built-in interaction assumptions from their original development environments (see Chapter 2). The existence of such assumptions is a violation of the intended separation of "core function" and "interconnection". In some cases difficulties associated with such assumptions can be resolved by *augmenting* components with appropriate caller and wrapper activities (see Section 5.2.3). In other cases, however, built-in assumptions might limit the flexibility of integrating a component in a given new application. For example, as we have seen in Section 6.4, it is difficult to efficiently reuse an executable program, which operates as a UNIX filter, in an interactive application.

Nevertheless, even with such "impure" components, the separation of activities and dependencies did provide valuable assistance in developing new applications from independently selected components. The overall experience from this separation has been very positive.

The positive experience from separating "core function" from "interconnection" in the "impure" world of today's components provides an indication that even more benefits are to be gained if we develop new programming languages with support for *"pure" components*. By "pure" components, we mean components with minimal assumptions about their interconnection patterns with the rest of the system. According to the results suggested by this thesis, such "pure" components would have maximum flexibility of reuse in new applications.

Although the form and nature of such components is left to future research, throughout this thesis we have informally defined minimality of interconnection assumptions to include at least the following four properties:

- Components interact with their environment through input and output resource ports only.

- Every input and output port of a component can be independently managed. That is, every input resource expected by a component can be independently produced and made accessible to the component. Likewise, every output resource can be independently made accessible to its consumers.

- Every input port can be connected to an arbitrary number of producers and every output port to an arbitrary number of consumers.

- Components make no assumptions about exclusive or shared ownership of resources. If necessary, coordination support for sharing resources should be defined completely outside the component.

The conventional programming language construct that more closely satisfies these properties is a sequential block of code which receives all its inputs and leaves all its outputs to independent local variables. In Chapter 5, we used activity augmentation by callers and wrappers to transform component types with stronger built-in interaction assumptions into approximations of this idealized construct. To facilitate the development of truly reusable software components, future programming languages should offer explicit support for abstractions with such properties.

8.4 Conclusion

Our ability to combine existing pieces of software to produce new applications holds the key to future increases in software productivity. However, after several years of extensive research efforts, systematic construction of large-scale software applications from existing parts remains an elusive goal.

This work has attempted to apply a coordination theory perspective to the representation and design of software systems, in order to explain why software reuse is such a hard problem, as well as to suggest ways for solving it. It has identified the inability of current programming languages to decouple ostensible function from interconnection details in software components. It has been able to come up with a novel way of decomposing software systems as sets of orthogonal subcomponents that separate the main functional pieces of an application from their patterns of interdependency. It has demonstrated that the patterns of interdependency, and the ways of managing them (coordination processes), can be systematized in a design framework of tractable complexity. Finally, it has shown

that a practical methodology for developing component-based applications can be based on the representations and frameworks introduced by the thesis.

Much remains to be done, but it is hoped that the initial results of my thesis are a step in the right direction. A better understanding of how software components interconnect to form complex systems will not only allow us to understand how to design components for reusability, but might also enable us to discover new software organizations-organizations in which software and hardware work together in as yet unimagined ways.

Appendix A

SYNTHESIS Coordination Code Listings

This appendix contains the coordination code generated by SYNTHESIS in order to create the example applications presented in Chapter 6. In order to make the code more readable, automatically generated variable names have been replaced with meaningful names and comments have been inserted where deemed appropriate.

A.1 Experiment 1: A Simple File Viewer

Implementation 1: Data Transfer using DDE

a. Application Entry Point

```
Sub Main ()

  start_exe0 "exe1.exe"        ' start C executable
  start_exe0 "exe2.exe"        ' start VB executable

End Sub
```

b. C coordination code (packaged in exe1.exe)

```
void init_C();
void init_DDE();

void main()
{
  init_C();

  /* Start user interface component */

  select_files();
}


/* Initialization function for C executable */

void init_C();
{
  /* init_DDE supplied by the coord. process library */
  init_DDE("VB", "RETR");
}


/* This function is called from within
 * select_files for each user-selected
 * code number
 */

void view_selected_files( int code )
{

  char str[20];

  /* Convert to string */
  itoa(str, code);

  /* Send str through a DDE call
   * with service name = VB
   * and  topic name = RETR
   *
   * dde_call supplied by coord. proc. library
   */
  dde_call( "VB", "RETR", str );
}
```

c. Visual Basic coordination code (packaged in exe2.exe)

```
Sub Main ()

  Init_VB

End Sub


' Initialization subroutine for Visual Basic
Sub Init_VB ()

  start_exe0 "\applic\msoffice\winword\winword.exe"

  init_DB

  init_DDE "VB", "RETR"

End Sub


Sub DDE_Execute(s As String)

  Dim code As Integer
  Dim filename As String

  code = CInt(s)        ' convert to integer

  filename = retrieve_filename( code )

  ' Send key sequence CTRL-O + filename + newline to window
  ' "Microsoft Word"
  gui_sendkeys1 "Microsoft Word", "^O@1~", filename

End Sub
```

Implementation 2: Data transfer using a shared file and semaphore synchronization

a. Application Entry Point

```
Sub Main ()

  start_exe0 "exe1.exe"        ' start C executable
  start_exe0 "exe2.exe"        ' start VB executable

End Sub
```

b. C coordination code (packaged in exe1.exe)

```
#include <stdio.h>

void init_C();

void main()
{
  init_C();

  /* Start user interface component */

  select_files();
}


/* Initialization function for C executable */
```

```
void init_C();
{

 /* Initialize semaphores to be used
  * for implementing lockstep prerequisite
  * synchronization with visual basic
  * executable
  */
  semaphore_init("Sema1", 0);
  semaphore_init("Sema2", 1);
}


/* This function is called from within
 * select_files for each user-selected
 * code number
 */

void view_selected_files( int code )
{

  char str[20];
  FILE *fd;

  /* Convert to string */
  itoa(str, code);

  /* Send data through shared file "File1"
   * use semaphores "Sema1" and "Sema2" to
   * implement lockstep prerequisite protocol
   */
  semaphore_P("Sema2");

  fd = fopen("File1", "w");

  fprintf( fd, "%s\n", str );

  fclose( fd );

  semaphore_V("Sema1");

}
```

c. Visual Basic coordination code (packaged in exe2.exe)

```
Global db_opened As Integer
Global viewer_started As Integer

Sub Main ()

  Init_VB

  retrieve_loop

End Sub


' Initialization subroutine for Visual Basic
Sub Init_VB ()

  ' Initialize variables used in persistent prerequisite
  ' coordination
  db_opened = False
  viewer_started = False

End Sub
```

```
Sub retrieve_loop ()

Dim fn As Integer
Dim s As String
Dim code As Integer
Dim filename As String

:11

   ' Implement persistent prereq 1 by checking
   ' if precedent has been called (and calling it if necessary)
   ' before each execution of the consequent

   If db_opened Then GoTo 12

   Open_DB
   db_opened = True

:12

   semaphore_P "Sema1"

   fn = FreeFile
   Open "File1" For Input As fn

   Input #fn, s

   Close #fn

   semaphore_V "Sema2"

   code = CInt(s)       'convert back to integer

   filename = retrieve_filename( code )

   ' Implement persistent prereq 2 as explained above

   If viewer_started Then GoTo 13

   start_exe0 "\applic\msoffice\winword\winword.exe"
   viewer_started = True

:13

   gui_sendkeys1 "Microsoft Word", "^O@1~", filename

   ' Current architecture does not specify termination
   ' condition. This subroutine loops indefinitely !!

   Goto 11

End Sub
```

A.2 Experiment 2: Key Word In Context

Implementation 1: Filter components interconnected using pipes

```
void kwic_main();

main()
  {

  kwic_main();

  }

void kwic_main()
  {
  /* Write and Read descriptors for the
   * three pipes used
   */
  int pipe1_w, pipe1_r;
  int pipe2_w, pipe2_r;
  int pipe3_w, pipe3_r;

  int old_t1, new_t1;
  int old_t2, new_t2;
  int old_t3, new_t3;

  /* Thread #1 */

  /* Returns pipe descriptors in
   * pipe1_w, pipe1_r
   */
  create_pipe(&pipe1_w, &pipe1_r);

  /* Forks a new UNIX thread.
   * Sets old_t1 = 1 in old thread.
   * Sets new_t1 = 1 in new thread.
   */
  fork_thread(&old_t1, &new_t1);

  if (new_t1) goto 11;

  gen_input( pipe1_w );
  goto 199;

11:

  /* Thread #2 */

  create_pipe(&pipe2_w, &pipe2_r);

  fork_thread(&old_t2, &new_t2);

  if (new_t2) goto 12;

  shift_lines( pipe1_r, pipe2_w );
  goto 199;

12:

  /* Thread #3 */

  create_pipe(&pipe3_w, &pipe3_r);

  fork_thread(&old_t3, &new_t3);

  if (new_t3) goto 13;

  sort_lines( pipe2_r, pipe3_w );
  goto 199;

13:

  /* Thread #4 */

  display_output( pipe3_r );

199:

}
```

Implementation 2: Filter components organized in main program-subroutine architecture

```
#include <sys/file.h>

void kwic_main();

main ()
  {

  kwic_main();

  }

void kwic_main()
  {

  /* File descriptors */
  int fd1;
  int fd2;
  int fd3;
  int fd4;
  int fd5;
  int fd6;

  fd1 = open("File1", O_WRONLY);

  gen_input( fd1 );

  close( fd1 );

  fd2 = open("File1", O_RDONLY);
  fd3 = open("File2", O_WRONLY);

  shift_lines( fd2, fd3 );

  close( fd2 );
  close( fd3 );

  fd4 = open("File2", O_RDONLY);
  fd5 = open("File3", O_WRONLY);

  sort_lines( fd4, fd5 );

  close( fd4 );
  close( fd5 );

  fd6 = open("File3", O_RDONLY);

  display_output( fd6 );

  close( fd6 );

  }
```

Implementation 3: Filter components organized in implicit invocation architecture

As explained in Section 6.3.2, this implementation introduces a redundant layer of coordination and is unnecessarily complex and inefficient. However, it is included here for completeness. Figure A.1 shows the overall organization of this implementation, and can help understand the code listing that follows.

```
void kwic_init();
void kwic_main();

main()
  {

  kwic_init();
  kwic_main();

  }

/* Semaphores for flow 1 */
int sema1, sema2;
/* Semaphores for flow 2 */
int sema3, sema4;
/* Semaphores for flow 3 */
int sema5, sema6;

/* Carrier resource for flow 1 */
char carrier1;
/* Carrier resource for flow 2 */
char carrier2;
/* Carrier resource for flow 3 */
char carrier3;

/* Write and Read descriptors for the
 * six pipes used
 */
int pipe1_w, pipe1_r;
int pipe2_w, pipe2_r;
int pipe3_w, pipe3_r;
int pipe4_w, pipe4_r;
int pipe5_w, pipe5_r;
int pipe6_w, pipe6_r;

/* Global variables used to
 * implement shared events
 * set to 1 to signal event
 */
int ev1;
int ev2;
int ev3;
int ev4;
int ev5;
int ev6;


void kwic_init()
  {

  /* Initialize semaphore pairs, as
   * required by the lockstep
   * prerequisite coordination
   * process, see Figure 4-11
   */
  semaphore_init(sema1, 0);
  semaphore_init(sema2, 1);
  semaphore_init(sema3, 0);
  semaphore_init(sema4, 1);
  semaphore_init(sema5, 0);
  semaphore_init(sema6, 1);

  }


void kwic_main()
  {

  /* Local variables where results
   * of fork_thread are stored
   */
  int old_t1, new_t1;
  int old_t2, new_t2;
  int old_t3, new_t3;
  int old_t4, new_t4;
  int old_t5, new_t5;
  int old_t6, new_t6;
  int old_t7, new_t7;
  int old_t8, new_t8;
  int old_t9, new_t9;

  /* Local variables where results
   * of check_event are stored
   */
  int test1;
  int test2;
  int test3;
  int test4;
  int test5;
  int test6;

  /* Local variables where results
   * of reading carrier resources
   * are stored
   */
  char c1;
  char c2;
  char c3;
```

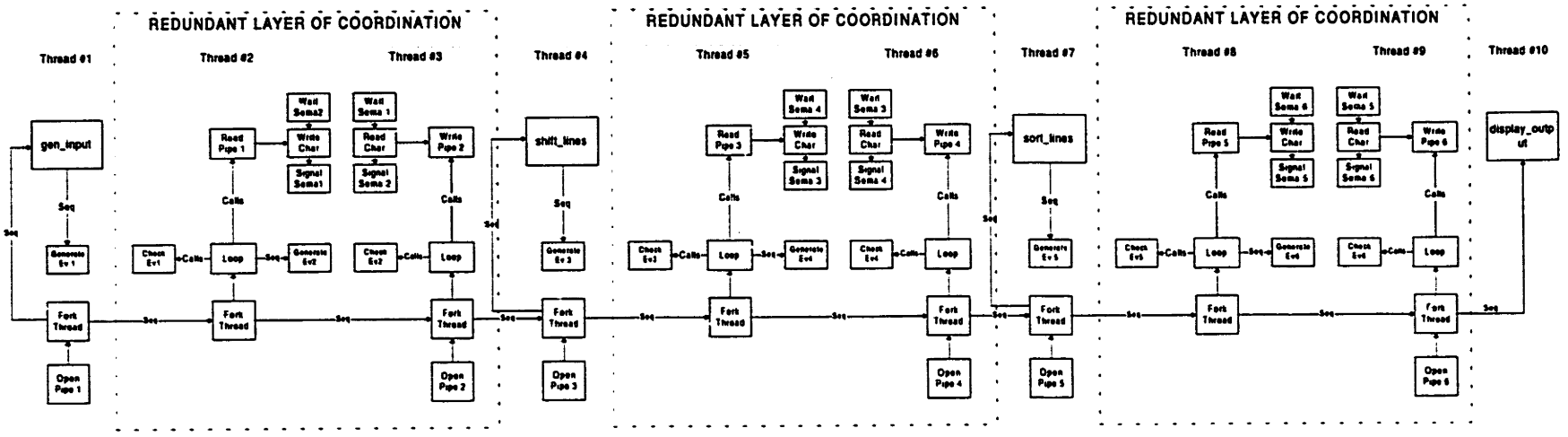


Figure A.1: Implementation 3 (Pipe components in implicit invocation organization)

```
  /* Thread #1 */

  /* Returns pipe descriptors in
   * pipe1_w, pipe1_r
   */
  create_pipe(&pipe1_w, &pipe1_r);

  /* Forks a new UNIX thread.
   * Sets old_t1 = 1 in old thread.
   * Sets new_t1 = 1 in new thread.
   */
  fork_thread(&old_t1, &new_t1);

  if (new_t1) goto 11;

  gen_input( pipe1_w );

  /* Signal end of input */
  generate_event(ev1);

  goto 199;

11:

  /* Thread #2 */

  fork_thread(&old_t2, &new_t2);

  if (new_t2) goto 13;

111:

  /* Loop here until thread #1 generates
   * event ev1 to signal no more input
   */

  /* Read pipe 1 */
  read( pipe1_r, &c1, 1);

  /* Write to carrier1 */
  semaphore_P(sema2);

  carrier1 = c1;

  semaphore_V(sema1);

  test1 = check_event( ev1 );

  if (!test1) goto 111;

  /* Signal no more input */
  generate_event( ev2 );

  goto 199;

12:

  /* Thread #3 */

  create_pipe(&pipe2_w, &pipe2_r);

  fork_thread(&old_t3, &new_t3);

  if (new_t3) goto 13;

121:

  /* Loop here until thread #2 generates
   * event ev2 to signal no more input
   */

  /* Read from carrier 1 */

  semaphore_P(sema1);

  c2 = carrier1;

  semaphore_V(sema2);

  /* Write to pipe 2 */

  write( pipe2_w, &c2, 1);

  test2 = check_event( ev2 );

  if (!test2) goto 121;

  goto 199;

13:

  /* Thread #4 */

  create_pipe(&pipe3_w, &pipe3_r);

  fork_thread(&old_t4, &new_t4);

  if (new_t4) goto 14;

  shift_lines( pipe2_r, pipe3_w );

  /* Signal no more shifted lines */
  generate_event( ev3 );

  goto 199;

14:

  /* Thread #5 */

  fork_thread(&old_t5, &new_t5);

  if (new_t5) goto 15;

141:

  /* Loop here until thread #4 generates
   * event ev3 to signal no more lines
   */

  /* Read pipe 3 */
  read( pipe3_r, &c3, 1);

  /* Write to carrier2 */
  semaphore_P(sema4);

  carrier2 = c3;

  semaphore_V(sema3);

  test3 = check_event( ev3 );

  if (!test3) goto 141;

  /* Signal no more input */
  generate_event( ev4 );

  goto 199;

15:

  /* Thread #6 */

  create_pipe(&pipe4_w, &pipe4_r);

  fork_thread(&old_t6, &new_t6);

  if (new_t6) goto 16;

151:
```

```
    /* Loop here until thread #5 generates
     * event ev4 to signal no more lines
     */

    /* Read carrier 2 */

    semaphore_P(sema3);

    c4 = carrier2;

    semaphore_V(sema4);

    /* Write pipe 4 */

    write( pipe4_w, &c4, 1);

    test4 = check_event( ev4 );

    if (!test4) goto 151;

    goto 199;

16:

    /* Thread #7 */

    create_pipe(&pipe5_w, &pipe5_r);

    fork_thread(&old_t7, &new_t7);

    if (new_t7) goto 17;

    sort_lines( pipe4_r, pipe5_w );

    /* Signal no more sorted lines */
    generate_event( ev5 );

    goto 199;

17:

    /* Thread #8 */

    fork_thread(&old_t8, &new_t8);

    if (new_t8) goto 18;

171:

    /* Loop here until thread #7 generates
     * event ev5 to signal no more input
     */

    /* Read pipe 5 */
    read( pipe5_r, &c1, 1);

    /* Write to carrier3 */
    semaphore_P(sema6);

    carrier3 = c5;

    semaphore_V(sema5);

    test5 = check_event( ev5 );

    if (!test5) goto 171;

    /* Signal no more input */
    generate_event( ev6 );

    goto 199;

18:

    /* Thread #9 */

    create_pipe(&pipe6_w, &pipe6_r);

    fork_thread(&old_t8, &new_t8);

    if (new_t8) goto 19;

181:

    /* Loop here until thread #8 generates
     * event ev6 to signal no more input
     */

    /* Read from carrier 3 */

    semaphore_P(sema5);

    c6 = carrier3;

    semaphore_V(sema6);

    /* Write to pipe 6 */

    write( pipe6_w, &c6, 1);

    test6 = check_event( ev6 );

    if (!test6) goto 181;

    goto 199;

19:

    /* Thread #10 */

    display_output( pipe6_r );

199:

}
```

Implementation 4: Server components interconnected using pipes

a. Automatically generated coordination code

```
/* Write and Read pipe descriptors */
int pipe1_w, pipe1_r;
int pipe2_w, pipe2_r;
int pipe3_w, pipe3_r;

/* Global variables used to signal
 * shared events.
 * Set to 1 by generate_event
 */
int ev1;
int ev2;

void kwic_main();

main()
  {

  kwic_main();

  }

void kwic_main()
  {

  int old_t1, new_t1;
  int old_t2, new_t2;
  int old_t3, new_t3;

  char line1[256];
  char line2[256];

  int test1;
  int test2;


  /* Thread #1 (Generate Text) */

  create_pipe(&pipe1_w, &pipe1_r);

  fork_thread(&old_t1, &new_t1);

  if (new_t1) goto 11;

  gen_input();

  /* Signal end of input to thread #2 */
  generate_event(ev1);

  goto 199;

11:

  /* Thread #2 (Shift Lines) */

  create_pipe(&pipe2_w, &pipe2_r);

  fork_thread(&old_t2, &new_t2);

  if (new_t2) goto 12;

111:

  /* Loop here until thread #1 generates
   * event ev1 to signal end of input
   * stream
   */
  convert_from_byte_stream( pipe1_r,
line1 );
  shift_line( line1 );

  /* Sets test1 to 1 if ev1 != 0 */
  test1 = check_event( ev1 );

  if (!test1) goto 111;

  /* Signal no more lines to thread #3
   */
  generate_event(ev2);

  goto 199;

12:

  /* Thread #3 (Sort Lines) */

  create_pipe(&pipe3_w, &pipe3_r);

  fork_thread(&old_t3, &new_t3);

  if (new_t3) goto 13;

 121:

  /* Loop here until thread #2 generates
   * event ev2 to signal no more lines
   */

  convert_from_byte_stream( pipe2_r,
line2 );

  next_line_to_sort( line2 );

  test2 = check_event( ev2 );

  if (!test2) goto 121;

  /* All lines have been read. Sort
   * them! */
  do_sort();

  goto 199;

13:

  /* Thread #4 (Display Output) */
  display_output();

199:

}

/* Called by gen_input for each new
 * input line */
void next_input_line(char *line)
  {
  convert_to_byte_stream( pipe1_w, line
);
  }

/* Called by USER-WRITTEN
 * convert_to_byte_stream
 * for each byte of a line
 */
void write_next_byte_in_stream(int pipe,
char byte)
  {
  write( pipe, byte, 1);
```

```
  }

/* Called by USER-WRITTEN
 * convert_from_byte_stream
 * to get next byte in stream
 */
char read_next_byte_in_stream(int pipe)
  {
  char c;

  read( pipe, &c, 1);

  return( c );
  }

/* Called by shift_line for each
 * circular shift of its input line
 */
void next_shifted_line(char *line)
  {
  convert_to_byte_stream( pipe2_w, line
);
  }

/* Called by do_sort for each final
 * sorted line */
void next_sorted_line( char *line)
  {
  convert_to_byte_stream( pipe3_w, line
);
  }

char rline[256];

/* Called by display_output to get
 * next line */
char *get_next_output_line()
  {
  convert_from_byte_stream( pipe3_r );

  return( rline );
  }
```

b. User-written coordination code

```
/* Convert a null-terminated string into
 * a newline-terminated stream of bytes
 *
 * stream is the stream descriptor
 * In UNIX it may be either a pipe or an
 * open file descriptor
 */

void convert_to_byte_stream(int stream,
char *line)
{

  char c;
  char *p = line;

  while( c = *p++ )
    write_next_byte_in_stream(stream, c);

  write_next_byte_in_stream(stream,
'\n');
}

/* Convert a newline-terminated stream
 * of bytes
 * into a null-terminated string
 *
 * stream is a stream (pipe or file)
 * descriptor
 */

void convert_from_byte_stream(int stream,
char *line)
{

  char *p = line;
  char c;

  while( (c =
        read_next_byte_in_stream(stream)
         != '\n' )
    *p++ = c;

  *p = '\0';

}
```

Implementation 5: Server components organized in main program-subroutine architecture

```
#include <sys/file.h>

int fd1;
int fd2;

void kwic_main();

main()
  {

  kwic_main();

  }

void kwic_main()
  {

  gen_input();

  fd1 = open("File1", O_WRONLY);

  do_sort();

  close (fd1);

  fd2 = open("File1", O_RDONLY);

  display_output();

  close (fd2);

  }


/* Called by gen_input for each new
 * input line
 */
void next_input_line( char *line )
  {

  shift_line( line );

  }

/* Called by shift_line for each
 * circular shift of its input line
 */

void next_shifted_line( char *line )
  {

  next_line_to_sort( line );

  }

/* Called by do_sort for each final
 * sorted line
 */
void next_sorted_line( char *line )
  {

  write_line(fd1, line);

  }

char rline[256];

/* Called by display_output to get next
 * line
 */
char *get_next_output_line()
  {

  read_line(fd2, rline);

  return( rline );
  }
```

Implementation 6: Server components organized in implicit invocation architecture

```
void kwic_init();
void kwic_main();

main()
  {

  kwic_init();
  kwic_main();

  }

/* Semaphores for flow 1 */
int sema1, sema2;
/* Semaphores for flow 2 */
int sema3, sema4;
/* Semaphores for flow 3 */
int sema5, sema6;

/* Carrier resource for flow 1 */
char carrier1[256];
/* Carrier resource for flow 2 */
char carrier2[256];
/* Carrier resource for flow 3 */
char carrier3[256];

kwic_init()
  {

  /* Initialize semaphore pairs, as
   * required
   * by the lockstep prerequisite
   * coordination
   * process, see Figure 4-11
   */
  semaphore_init(sema1, 0);
  semaphore_init(sema2, 1);
  semaphore_init(sema3, 0);
  semaphore_init(sema4, 1);
  semaphore_init(sema5, 0);
  semaphore_init(sema6, 1);

  }

/* Write and Read pipe descriptors */
int pipe1_w, pipe1_r;
int pipe2_w, pipe2_r;
int pipe3_w, pipe3_r;

/* Global variables used to signal
 * shared events
 * Set to 1 by generate_event
 */
int ev1;
int ev2;

kwic_main()
  {

  int old_t1, new_t1;
  int old_t2, new_t2;
  int old_t3, new_t3;

  char line1[256];
  char line2[256];

  int test1;
  int test2;

  /* Thread #1 (Generate Text) */

  fork_thread(&old_t1, &new_t1);

  if (new_t1) goto 11;

  gen_input();

  /* Signal end of input */
  generate_event(ev1);

  goto 199;

11:
  /* Thread #2 (Shift Lines) */

  fork_thread(&old_t2, &new_t2);

  if (new_t2) goto 12;

111:

  /* Loop here until thread #1 generates
   * event
   * ev1 to signal no more input lines
   */
  semaphore_P(sema1);

  /* Read next input line from carrier
   * resource */
  strcpy(line1, carrier1);

  semaphore_V(sema2);

  shift_line( line1 );

  test1 = check_event( ev1 );

  if (!test1) goto 111;

  /* Signal end of shifted lines */
  generate_event(ev2);

  goto 199;

12:
  /* Thread #3 (Sort Lines) */

  fork_thread(&old_t3, &new_t3);

  if (new_t3) goto 13;

 121:

  /* Loop here until thread #2 generates
   * event
   * ev2 to signal no more shifted lines
   */
  semaphore_P(sema4);

  /* Read next shifted line from carrier
   * resource */
  strcpy(line2, carrier2);

  semaphore_V(sema3);

  next_line_to_sort( line2 );

  test2 = check_event( ev2 );

  if (!test2) goto 121;

  do_sort();
```

```
  goto 199;

13:

  /* Thread #4 (Display output) */
  display_output();

199:

}

void next_input_line(char *line)
  {

  semaphore_P(sema2);

  /* Write next input line to carrier
   * resource */
  strcpy(carrier1, line);

  semaphore_V(sema1);
  }

void next_shifted_line(char *line)
  {

  semaphore_P(sema4);

  /* Write next shifted line to carrier
   * resource */
  strcpy(carrier2, line);

  semaphore_V(sema3);
  }

void next_sorted_line( char *line)
  {
  semaphore_P(sema6);

  /* Write next sorted line to carrier
   * resource */
  strcpy(carrier3, line);

  semaphore_V(sema5);

  }
char rline[1024];

char *get_next_output_line()
  {

  semaphore_P(sema5);

  /* Read next output line from carrier
   * resource */
  strcpy( rline, carrier3 );

  semaphore_V(sema6);

  return( rline );
  }
```

Implementation 7: Mixed components interconnected using pipes

```
/* Write and Read pipe descriptors */
int pipe1_w, pipe1_r;
int pipe2_w, pipe2_r;
int pipe3_w, pipe3_r;

/* Global variables used to signal
 * shared events
 * Set to 1 by generate_event
 */
int ev1;
int ev2;

void kwic_main();

main()
  {

  kwic_main();

  }

void kwic_main()
  {

  int old_t1, new_t1;
  int old_t2, new_t2;
  int old_t3, new_t3;

  char line1[256];

  int test1;
  int test2;


  /* Thread #1 (Generate Text) */

  create_pipe(&pipe1_w, &pipe1_r);

  fork_thread(&old_t1, &new_t1);

  if (new_t1) goto l1;

  gen_input(pipe1_w);

  /* Signal end of input to thread #2 */
  generate_event(ev1);

  goto 199;

l1:

  /* Thread #2 (Shift Lines) */

  create_pipe(&pipe2_w, &pipe2_r);

  fork_thread(&old_t2, &new_t2);

  if (new_t2) goto l2;

l11:

  /* Loop here until thread #1 generates
   * event ev1 to signal end of input
   */
  convert_from_byte_stream( pipe1_r,
line1 );
  shift_line( line1 );

  /* Sets test1 to 1 if ev1 != 0 */
  test1 = check_event( ev1 );

  if (!test1) goto l11;

  /* Signal no more lines to thread #3
   */
  generate_event(ev2);

  goto 199;

l2:

  /* Thread #3 (Sort Lines) */

  create_pipe(&pipe3_w, &pipe3_r);

  fork_thread(&old_t3, &new_t3);

  if (new_t3) goto l3;

  sort_lines( pipe2_r, pipe3_w )

  goto 199;

l3:

  /* Thread #4 (Display Output) */
  display_output( );

199:

}


/* Called by USER-WRITTEN
 * convert_to_byte_stream
 * for each byte of a line
 */
void write_next_byte_in_stream(int pipe,
char byte)
  {
  write( pipe, byte, 1);
  }


/* Called by USER-WRITTEN
 * convert_from_byte_stream
 * to get next byte in stream
 */
char read_next_byte_in_stream(int pipe)
  {
  char c;

  read( pipe, &c, 1);

  return( c );
  }


/* Called by shift_line for each
 * circular shift
 * of its input line
 */
void next_shifted_line(char *line)
  {
  convert_to_byte_stream( pipe2_w, line
);
  }


char rline[256];

/* Called by display_output to get next
 * line
 */
```

```
char *get_next_output_line()
  {

  convert_from_byte_stream( pipe3_r,
rline );

  return( rline );
  }
```

Implementation 8: Mixed components organized in main program-subroutine architecture

```
#include <sys/file.h>

void kwic_main();

main ()
  {

  kwic_main

  }

void kwic_main()
  {

  /* File descriptors */
  int fd1;
  int fd2;
  int fd3;
  int fd4;
  int fd5;
  int fd6;

  char line1[256];

  fd1 = open("File1", O_WRONLY);

  /* gen_input is a filter and writes
   * all output into File1
   */
  gen_input( fd1 );

  close( fd1 );

  fd2 = open("File1", O_RDONLY);
  fd3 = open("File2", O_WRONLY);

111:

  /* Loop here until end of File1 */

  convert_from_byte_stream( pipe1_r,
line1 );
  shift_line( line1 );

  /* Sets test1 to 1 if end_of_file */
  test1 = end_of_file( fd2 );

  if (!test1) goto 111;

  close( fd2 );
  close( fd3 );

  fd4 = open("File2", O_RDONLY);
  fd5 = open("File3", O_WRONLY);

  /* sort_lines is a filter which reads
   * input from File2 and writes sorted
   * output into File3
   */
  sort_lines( fd4, fd5 );

  close( fd4 );
  close( fd5 );

  fd6 = open("File3", O_RDONLY);

  display_output( );

  close( fd6 );

  }


/* Called by USER-WRITTEN
 * convert_to_byte_stream
 * for each byte of a line
 */
void write_next_byte_in_stream(int pipe,
char byte)
  {
  write( pipe, byte, 1);
  }


/* Called by USER-WRITTEN
 * convert_from_byte_stream
 * to get next byte in stream
 */
char read_next_byte_in_stream(int pipe)
  {
  char c;

  read( pipe, &c, 1);

  return( c );
  }


/* Called by shift_line for each
 * circular shift
 * of its input line
 */
void next_shifted_line(char *line)
  {
  convert_to_byte_stream( fd3, line );
  }


char rline[256];

/* Called by display_output to get next
 * line
 */
char *get_next_output_line()
  {

  convert_from_byte_stream( fd6, rline );

  return( rline );
  }
```

Implementation 9: Mixed components organized in implicit invocation architecture

As explained in Section 6.3.2, this implementation introduces a redundant layer of coordination and is unnecessarily complex and inefficient. However, it is included here for completeness. Figure A.2 shows the overall organization of this implementation, and can help understand the code listing that follows.

```
/* Write and Read pipe descriptors */
int pipe1_w, pipe1_r;
int pipe2_w, pipe2_r;
int pipe3_w, pipe3_r;

/* Global variables used to signal
 * shared events
 * Set to 1 by generate_event
 */
int ev1;
int ev2;
int ev3;
int ev4;

/* Semaphores for flow 1 */
int sema1, sema2;
/* Semaphores for flow 2 */
int sema3, sema4;
/* Semaphores for flow 3 */
int sema5, sema6;

/* Carrier resource for flow 1 */
char carrier1;
/* Carrier resource for flow 2 */
char carrier2;
/* Carrier resource for flow 3 */
char carrier3;


void kwic_init();
void kwic_main();

main()
  {

  kwic_init();
  kwic_main();

  }


void kwic_init()
  {

  /* Initialize semaphore pairs, as
   * required
   * by the lockstep prerequisite
   * coordination
   * process, see Figure 4-11
   */
  semaphore_init(sema1, 0);
  semaphore_init(sema2, 1);
  semaphore_init(sema3, 0);
  semaphore_init(sema4, 1);
  semaphore_init(sema5, 0);
  semaphore_init(sema6, 1);

  }


void kwic_main()
  {

  int old_t1, new_t1;
  int old_t2, new_t2;
  int old_t3, new_t3;
  int old_t4, new_t4;
  int old_t5, new_t5;
  int old_t6, new_t6;

  char line1[256];

  char c1;
  char c2;
  char c3;

  int test1;
  int test2;
  int test3;
  int test4;


  /* Thread #1 (Generate Text) */

  create_pipe(&pipe1_w, &pipe1_r);

  fork_thread(&old_t1, &new_t1);

  if (new_t1) goto l1;

  gen_input(pipe1_w);

  /* Signal end of input to thread #2 */
  generate_event(ev1);

  goto l99;
```

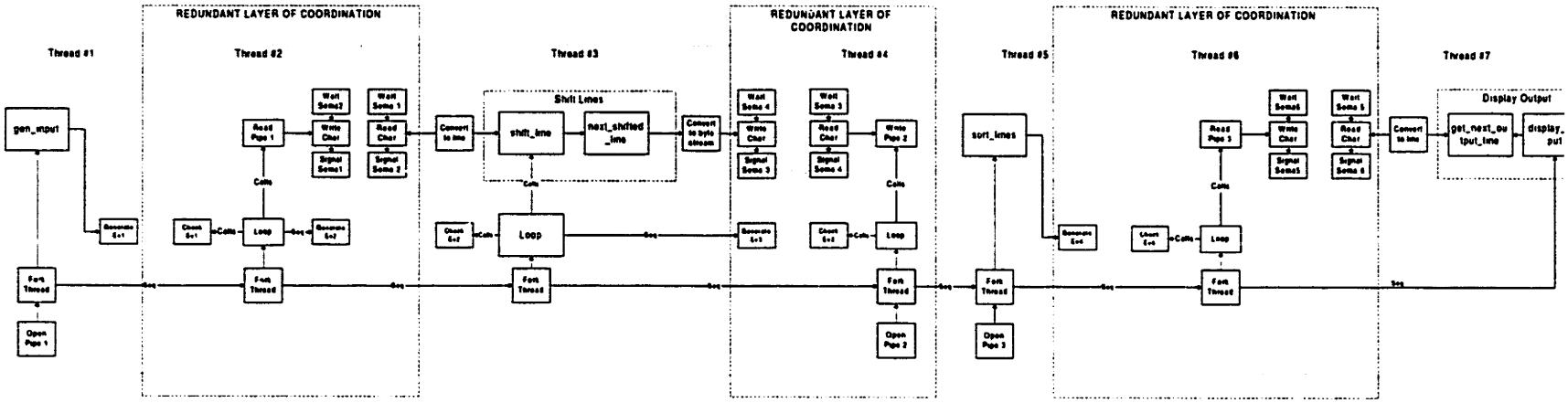


Figure A.2: Implementation 9 (Mixed components in implicit invocation organization)

```
11:

  /* Thread #2 */

  fork_thread(&old_t2, &new_t2);

  if (new_t2) goto 12;

111:

  /* Loop here until thread #1 signals
   * end of input
   */

  /* Read pipe 1 */

  read( pipe1_r, &c1, 1);

  /* Write carrier 1 */

  semaphore_P(sema2);

  carrier1 = c1;

  semaphore_V(sema1);

  test1 = check_event(ev1);

  if (!test1) goto 111;

  /* Signal end of input to thread #3 */
  generate_event(ev2);

  goto 199;

12:

  /* Thread #3 (Shift Lines) */

  fork_thread(&old_t3, &new_t3);

  if (new_t3) goto 13;

121:

  /* Loop here until thread #2 generates
   * event ev2 to signal end of input
   */
  convert_to_line( sema1, sema2,
&carrier1, line1 );
  shift_line( line1 );

  test2 = check_event( ev2 );

  if (!test2) goto 121;

  /* Signal no more lines to thread #4
   */
  generate_event(ev3);

  goto 199;

13:

  /* Thread #4 */

  create_pipe(&pipe2_w, &pipe2_r);

  fork_thread(&old_t4, &new_t4);

  if (new_t4) goto 14;

131:

  /* Loop here until thread #3 generates
   * event ev3 to signal end of input
   */

  /* Read carrier 2 */

  semaphore_P(sema3);

  c2 = carrier2;

  semaphore_V(sema4);

  /* Write Pipe 2 */

  write(pipe2_w, &c2, 1);

  test3 = check_event(ev3);

  if (!test3) goto 131;

  goto 199;

14:

  /* Thread #5 (Sort Lines) */

  create_pipe(&pipe3_w, &pipe3_r);

  fork_thread(&old_t5, &new_t5);

  if (new_t5) goto 15;

  sort_lines( pipe2_r, pipe3_w )

  generate_event(ev4);

  goto 199;

15:

  /* Thread #6 */

  fork_thread(&old_t6, &new_t6);

  if (new_t6) goto 16;

151:

  /* Loop here until thread #5 signals
   * no more sorted lines
   */


  /* Read pipe 3 */

  read( pipe3_r, &c3, 1);

  /* Write carrier 3 */

  semaphore_P(sema6);

  carrier3 = c3;

  semaphore_V(sema5);

  test4 = check_event(ev4);

  if (!test4) goto 151;

  goto 199;

16:

  /* Thread #7 (Display Output) */
  display_output( );

199:
```

```
}

/* Called by USER-WRITTEN
 * convert_from_line
 * for each byte of a line
 */
void write_next_byte_(int sema1, int
sema2, char *carrier, char byte)
  {
  semaphore_P(sema1);

  *carrier = byte;

  semaphore_V(sema2);
  }


/* Called by USER-WRITTEN
 * convert_from_byte_stream
 * to get next byte in stream
 */
char read_next_byte(int sema1, int sema2,
char *carrier)
  {
  char c;

  semaphore_P(sema1);

  c = *carrier;

  semaphore_V(sema2);

  return( c );
  }


/* Called by shift_line for each
 * circular shift
 * of its input line
 */
void next_shifted_line(char *line)
  {
  convert_from_line( sema4, sema3,
&carrier2, line );
  }


char rline[256];

/* Called by display_output to get next
 * line
 */
char *get_next_output_line()
  {

  convert_to_line( sema5, sema6,
&carrier3, rline );

  return( rline );
  }
```

b. *User-written coordination code*

```
/* Convert a null-terminated string into
 * a newline-terminated stream of bytes
 *
 *
 */

void convert_from_line(int sema1, int
sema2, char *carrier, char *line)
{

  char c;
  char *p = line;

  while( c = *p++ )
    write_next_byte(sema1, sema2,
carrier, c);

 write_next_byte(sema1, sema2, carrier,
'\n');
}

/* Convert a newline-terminated stream
 * of bytes
 * into a null-terminated string
 *
 */

void convert_to_line(int sema1, int
sema2, char *carrier, char *line)
{

  char *p = line;
  char c;

  while( (c =
        read_next_byte(sema1, sema2,
carrier)
          != '\n' )
    *p++ = c;

  *p = '\0';

}
```

A.3 Experiment 3: Interactive TEX System

a. Coordination code automatically generated by SYNTHESIS

```
Option Explicit

' Global variables used for implementing
' Perishable and Persistent Prerequisites

Global EditorStarted As Integer
Global ViewerStarted As Integer
Global PSValid As Integer
Global LatexDone As Integer

' Global variables used as carrier resources
' in flow dependencies

Global Current_TexFilename As String
Global Current_DviFilename As String
Global Current_PsFilename As String

' Fixed name of replica file used for
' managing Flow 1 (see Section 6.4.2)

Global Const WORD_REPLICA = "d:\tmp.tex"


Sub Main ()

    ' Initialization

    Init

    ' CONTROLLER

    tex_main_control

End Sub


Sub Init ()

    ' Initialize the global variables used to implement
    ' persistent and perishable prerequisites

    EditorStarted = False
    ViewerStarted = False
    LatexDone = False
    PSValid = False

End Sub


Sub new_document (filename As String)

    'Write filename to global variable

    Current_TexFilename = filename

    'Implement persistent prerequisite using consumer pull

    If EditorStarted Then GoTo editstarted

        ' START EDITOR

        ' start_exe0 is a coordination library component that
        ' starts the specified executable program
```

```
        start_exe0 "\applic\msoffice\winword\winword.exe"

        EditorStarted = True

    : editstarted


    ' Create a replica of filename to manage sharing
    ' of the .tex file between the editor (Word) and TeX
    FileCopy Current_TexFilename, WORD_REPLICA

    ' OPEN FILE

    ' gui_sendkeys is a coordination library component that
    ' formats and sends a key sequence to the specified window
    gui_sendkeys1 "Microsoft Word", "^O@1~", WORD_REPLICA

    ' DETECT FILE CHANGE

    detect_change

End Sub


Sub file_changed ()

    Dim f1 As String
    Dim f2 As String
    Dim f3 As String

    'Invalidate .dvi file
    LatexDone = False

    'Invalidate .ps file
    PSValid = False


    'Get current filename from global variable

    f1 = Current_TexFilename

    ' Write replica back to original .tex file

    FileCopy WORD_REPLICA, f1

    ' Convert filename to UNIX format expected by latex executable
    ' (replace backslashes in pathname with slashes)

    f2 = Convert_To_Latex(f1)

    ' CALL LateX

    ' call_exe1 is a coordination library component that
    ' starts an executable and waits until it completes

    Call_Exe1 "c:\emtex\latex.pif", "LaTeX", f2

    'Enable dvips (Perishable Flow)

    LatexDone = True

    'Get .dvi filename
    f3 = Convert_To_dvi(f2)

    ' Write to global variable

    Current_DviFilename = f3

    'Check if dvi driver has started

    If ViewerStarted Then GoTo dvistarted

        ' START VIEWER

        ' start_exe0 is a coordination library component that
        ' starts the specified executable program
```

```
        start_exe1 "\tex\dviwin\dviwin.exe", "-1"

        ViewerStarted = True

: dvistarted

    'Refresh Dvi driver

    start_exe2 "\tex\dviwin\dviwin", "-1", f3

End Sub


Sub print_document (printer As String)

    Dim f1 As String
    Dim f2 As String
    Dim f3 As String

    'If postscript file is not current

    If PSValid Then GoTo canprint

        'Wait until LaTeX is done

        wait_for_event (LatexDone)

        'Read latest .dvi filename

        f1 = Current_DviFilename

        'Convert to postscript

        Call_Exe1 "c:\emtex\dvips.pif", "Dvips", f1

        PSValid = True

        f2 = Convert_To_ps(f1)

        Current_PsFilename = f2

: canprint

    'Send postscript to specified printer

    f3 = Current_PsFilename

    ' DOS command:
    '     copy <filename> <printer_device_name>
    ' sends a postscript file to a postscript printer
    ' without further processing

    start_exe3 "c:\windows\dosprmpt.pif", "copy", f3, printer


End Sub


Sub end_application ()

    'Quit Dviwin

    start_exe1 "c:\tex\dviwin\dviwin", "-1 -c"


    'Quit Microsoft Word

    gui_sendkeys0 "MicrosoftWord", "%{FX}"


End Sub
```

b. User-Written Coordination Code

```
Function Convert_To_Latex (s As String) As String

    ' The DOS version of TeX (emtex) expects pathnames
    ' separated by slashes, rather than backslashes.
    '
    ' Also, it does not understand dos drive numbers

    ' This function converts a dos pathname to the
    ' equivalent unix pathname

    'Replace all \'s by /

    Dim f As String
    f = ""
    Dim i As Integer
    For i = 1 To Len(s)
        Dim c As String
        c = Mid$(s, i, 1)
        If c = "\" Then f = f & "/" Else f = f & c
    Next i

    'Wipe suffix, if any

    Dim p As Integer

    Dim root As String

    p = Len(f)

    If Mid$(f, p - 3, 1) = "." Then
        root = Left$(f, p - 4)
    Else
        root = f
    End If

    'Wipe out drive number, if any
    If Len(root) >= 2 And Mid$(root, 2, 1) = ":" Then
        root = Mid$(root, 3)
    End If

    Convert_To_Latex = root

End Function
```

A.4 Experiment 4: A Collaborative Editor Toolkit

Coordination code automatically generated by SYNTHESIS

```
#include <string.h>

/* Global variable defined in each participant
 * Set to 1 if another participant has acquired master status
 * Set to 0 otherwise
 */
int master_exists;

/* Handler called whenever a participant starts or tries to
 * load a file
 */
void read_file(char *filename)
{

  /* Broadcast intention to load file to all
   * participants. Recipients respond by saving
   * their buffer contents.
   *
   * Call does not return before all recipients
   * have responded
   */
  DDE_Broadcast0("memNEW");

  /* Now load file */
  readin(filename, 0);

}


/* Handler called whenever a participant presses the
 * "acquire master" key sequence
 */
void master_acquire()
{

  /* If no other master, acquire master status */

  if (master_exists) goto l1;

  m_acq();

l1:

}


/* Handler called whenever a participant presses the
 * "release master" key sequence
 */
void master_release()
{

  /* If no other master, release master status */

  if (master_exists) goto l2;

  m_rel();

l2:

}

/* Handler called whenever a participant presses the
 * "quit editor" key sequence
 */
void quit_editor()
```

```
{

  /* If no other master, release master status
   * before quitting
   */
  if (master_exists) goto 13;

  m_rel();

13:

  quit(0, 0);

}


/* Called from within the editor event loop,
 * whenever a local event has been detected
 */

void local_event(int event)
{

  /* If no other master, send local event */

  if (master_exists) goto 14;

  m_event(event);

14:

}

/* Called from within the editor event loop,
 * in order to wait for global events
 */

int global_event()
{

  int event;

  /* Wait until a global event has arrived
   * global_in_check and global_in_get
   * are coordination library routines for
   * accessing the global event queue
   * used as part of managing the cumulative
   * many-to-many event flow
   */

15:
  if (global_in_check()) goto 16;

  WaitMessage();

  goto 15;

16:
  event = global_in_get();

  return( global_in_get() );
}


/* Called from m_acq */

void master_acquired()
{

        /* Broadcast master acquisition */

        DDE_Broadcast0("memMACQ");
}


/* Called from m_rel */
```

```
void master_released ()
{

        /* Broadcast master release */

        DDE_Broadcast0("memMREL");

}

/* Called from m_event */

void send_local_event(int event)
{

        char cevent[20];

        /* Convert to string */
        itoa(cevent, event);

        /* Broadcast local event */
        DDE_Broadcast1("memCHAR, cevent);

}



/* DDE_Broadcast_Handler is defined at each participant editor
 * It is called from inside the DDE Callback function
 * whenever a WILDCONNECT (broadcast) transaction is detected.
 */

DDE_Broadcast_Handler(char *service, int fSameInst)
/* service     is the transaction service name */
/* fSameInst   is 1 if we are the same participant who sent the transaction */
{

        char *b1;
        char *cmd;
        char *arg;
        int event;

        b1 = strchr( service , "|" );

        /* All messages that concern me have the format :

                    cmd|arg

          */

        If (b1 == (char *)NULL ) goto 110;

        *b1 = '\0';
        cmd = service;
        arg =  (b1 + 1);


        /* if no other master exists and a new instance wants to load a file,
          * save current buffer contents
          */

        if ( !strcmp(cmd, "memNEW") ) goto 111;

        if (master_exists) goto 112;

        /* Save file */
        filesave(0, 0);
112:
111:

        /* if I received a new character, enter into my input buffer */

        if ( !strcmp( cmd, "memCHAR") ) goto 113;

         /* Convert to integer */
        event = atoi(arg);
```

```
        global_in_put(event);

113:

        /* if I received a Master acquire, then set master_exists, unless I am the
          *  one who is sending it */

        if ( !strcmp( cmd, "memMACQ") ) goto 114;

         /* Check if I am the one sending the transaction */
        if (!fSameInst) goto 115;

         master_exists = 1;

115:
114:

        /* if I received a Master release then no other master exists */

        if ( !strcmp( cmd, "memMREL") ) goto 116;

        master_exists = 0;

116:
110:

}
```

References

[Adler95] Richard M. Adler. Emerging Standards for Component Software. *IEEE Computer*, March 1995, pp. 68-77.

[Alexander77] C. Alexander, et. al. *A Pattern Language*. Oxford University Press, New York, 1977.

[Allen84] James F. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence* 23 (1984), pp. 123-154.

[Allen89] B.P. Allen and S.D. Lee. A Knowledge-Based Environment for the Development of Software Parts Composition Systems. In *Proceedings, 12th Int'l Conference on Software Engineering*, 1989, pp. 104-112.

[Allen94] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings, 16th Int'l Conference on Software Engineering*, 1994.

[Andreoli94] J.M. Andreoli, H. Gallaire and R. Pareschi. Rule-Based Object Coordination. In *Proceedings, ECOOP'94 Workshop on Coordination*, 1994.

[Andrews91] Gregory R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, Vol. 23, No. 1, March 1991, pp. 49-90.

[Apple93] Apple Computer. *Inside Macintosh Volume 7: Interapplication Communication*. Addison-Wesley, 1993.

[Apple94] Apple Computer. *AppleScript Language Manual, English Dialect.* Addison-Wesley, 1994.

[Bacon93] Jean Bacon. *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems*. Addison-Wesley Publishing Company, 1993.

[Beach92] Brian W. Beach. Connecting Software Components with Declarative Glue. In *Proceedings, 14th Int'l Conference on Software Engineering*, 1992, pp. 120-137.

[Bell71] C. Gordon Bell and Allen Newell. *Computer Structures: Readings and Examples*. McGraw-Hill, New York, 1971.

[Biggerstaff89] T. J. Biggerstaff and A. J. Perlis. *Software Reusability*. Volumes 1 and 2, ACM Press/Addison Wesley, 1989.

[Birman89] K. Birman, R. Cooper, T. Joseph, K. Kane, and F. Schmuck. *The ISIS System Manual*, June 19, 1989.

[Boehm88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, May 1988, pp. 61-72.

[Boyle84] J. M. Boyle and M. N. Muralidharan. Program Reusability through Program Transformation. *IEEE Transactions on Software Engineering*, September 1984, pp. 574-588.

[Callahan91] J. R. Callahan and J. M. Purtilo. A Packaging System For Heterogeneous Execution Environments. *IEEE Transactions on Software Engineering*, Vol. 17, No. 6, June 1991, pp 626-635.

[Cameron89] John R. Cameron. *JSP and JSD: The Jackson Approach to Software Development* (Second Edition). IEEE Computer Society Press, 1989.

[Carriero89] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, Vol. 21, No. 3, September 1989, pp. 324-357.

[Dellarocas94] C. Dellarocas, J. Lee, T. W. Malone, K. Crowston and B. Pentland. Using a Process Handbook to Design Organizational Processes. In *Proceedings, AAAI Spring Symposium on Computational Organization Design*, March 21-23, 1994, Stanford, CA, pp. 50-56.

[DeRemer75] Frank DeRemer and Hans H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2, June 1976, pp. 80-86.

[DoD83] U.S. Department of Defense. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A, January 1983.

[Dongarra87] J. J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, Vol. 30, No. 5, May 1987, pp. 403-407.

[Gamma93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Abstractions and reuse of object-oriented design. In *Proceedings, ECOOP '93*, Springer Verlag LNCS 707, July 1993.

[Gamma94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Micro-Architectures for Reusable Object-Oriented Design*. Addison-Wesley, 1994.

[Garlan88] D. Garlan, G. E. Kaiser, and D. Notkin. On the Criteria to be Used in Composing Tools into Systems. Technical Report 88-08-09, Department of Computer Science, University of Washington, August 1988.

[Garlan94a] David Garlan and Mary Shaw. An Introduction To Software Architecture. Technical Report CMU-CS-94-166, January 1994. Also appears as CMU/SEI-94-TR-21, ESC-TR-84-21.

[Garlan94b] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. *Proceedings, ACM SIGSOFT '94 Symposioum on Foundations of Software Engineering*, New Orleans, LA, December 1994.

[Garlan95] D. Garlan, R. Allen and J. Ockerbloom. Architectural Mismatch or Why it's hard to build systems out of existing parts. In *Proceedings, 17th International Conference on Software Engineering*, Seattle WA, April 1995.

[Gelernter92] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, Vol. 35, No. 2, February 1992, pp. 97-107.

[Gibbons87] Phillip B. Gibbons. A Stub Generator for Multilanguage RPC in Heterogeneous Environments. *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, January 1987, pp. 77-87.

[Gogouen86] Joseph A. Goguen. Reusing and Interconnecting Software Components. *IEEE Computer*, February 1986, pp. 16-28.

[Hayes87] R. Hayes and R.D. Schlichting. Facilitating Mixed Language Programming in Distributed Systems. *IEEE Transactions on Software Engineering*, Vol. 13, No. 12, December 1987, pp.1254-1264.

[Hoare85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, U.K. Ltd., 1985.

[IMSL87] *IMSL Math/Library User's Manual.* 1.0 Edition, Houston, Texas, 1987.

[Inmos84] Inmos Ltd. *Occam Programming Manual.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[Intellicorp92] *Kappa-PC Reference Manual, Version 2.0.* Intellicorp, Inc., Mountain View, CA, 1992.

[Jones84] T. Capers Jones. Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, September 1984, pp. 488-494.

[Jones85] M.B. Jones, R.F. Rashid and M.R. Thompson. Matchmaker: An Interface Specification Language for Distributed Processing. In *Proceedings, 12th ACM Symposium of Principles of Programming Languages*, 1985, pp. 225-235.

[Katz87] S. Katz, C. A. Richter, and K. The. PARIS: A system for reusing partially interpreted schemas. In *Proceedings, 9th International Conference on Software Engineering* (Monterey, CA), IEEE Computer Society Press, Los Alamitos, CA, pp. 377-385.

[Knister90] M. J. Knister and A. Prakash. DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors. In *Proceedings, CSCW 90*, Los Angeles, CA, October 1990, pp. 343-355.

[Knuth89] Donald Knuth. *The TEXbook.* Addison Wesley, Fifteenth edition, 1989.

[Kogut94a] Paul Kogut and Kurt Wallnau. Software Architecture and Reuse: Senses and Trends. WADAS 94 Tutorial Notes. 1994.

[Kogut94b] Paul Kogut and Paul Clements. Features of Architecture Representation Languages. Carnegie Mellon University Technical Report CMU/SEI. Number to be assigned. Draft of December 1994.

[Krasner88] G. Krasner and S. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming* 1, 3, August-September 1988, pp. 26-49.

[Krueger92] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, Vol. 24, No. 2, June 1992, pp. 131-183.

[Lam94] S. S. Lam and A. U. Shankar. A Theory of Interfaces and Modules I-Composition Theorem. *IEEE Transactions on Software Engineering*, Vol. 20, No. 1, January 1994, pp. 55-71

[Lamb87] David Alex Lamb. IDL: Sharing Intermediate Representations. *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, July 1987, pp. 297-318.

[Lane90] Thomas G. Lane. A Design Space and Design Rules for User Interface Software Architecture. Technical Report CMU/SEI-90-TR-22, ESD-90-TR-223, November 1990.

[Lawrence89] D. M. Lawrence and B. Straight. *MicroEmacs Full Screen Text Editor Reference Manual, version 3.10*, March 1989.

[LeBlanc85] T. J. LeBlanc and S. A. Friedberg. HPC: A Model of Structure and Change in Distributed Systems. *IEEE Transactions on Computers*, Vol. C-34, No. 12, December 1985, pp. 1114-1129.

[Liang90] L. Liang, S.T. Chanson and G.W. Neufeld. Process Groups and Group Communications: Classifications and Requirements. *IEEE Computer*, February 1990, pp. 56-65

[Lucco90] S. Lucco and O. Sharp. Delirium: An Embedding Coordination Language. In *Proceedings, Supercomputing '90*, IEEE Computer Society Press, pp. 515-524.

[Magee89] J. Magee, J. Kramer and M. Sloman. Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, Vol. 15, No. 6, June 1989, pp. 663-675.

[Malone93] T.W. Malone, K. Crowston, J. Lee and B. Pentland. Tools for Inventing Organizations: Toward a Handbook of Organizational Processes, In *Proceedings, 2nd IEEE Workshop on Enabling Tech. Infrastructure for Collaborative Enterprises*, April 20-22, 1993.

[Malone94] Thomas W. Malone and Kevin Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, Vol. 26, No. 1, March 1994, pp. 87-119.

[Mettala92] E. Mettala and M. H. Graham. The domain-specific software architecture program. Tech. Report CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, June 1992.

[Meyers89] S. Meyers and S.P. Reiss. Representing Programs in Multiparadigm Software Development Environments. In *Proceedings, 13th Int'l Computer Science and Applications Conf.*, 1989 (COMPSAC-89), pp. 420-427.

[Microsoft93] *Language Reference, Microsoft Visual Basic Version 3.0.* Microsoft Corporation, Redmont, WA. 1993.

[Microsoft94] *OLE2 Programmers' Reference*, Vols.1 and 2, Microsoft Press, Redmond, Wash., 1994.

[Muhlhauser93] M. Muhlhauser, W. Gerteis and L. Heuser. DOCASE: A Methodic Approach to Distributed Programming. *Communications of the ACM*, Vol. 36, No. 9, September 1993, pp.127-138.

[Nierstrasz94] Oscar Nierstrasz. Composing Active Objects. In G.Agha, P.Wegner and A.Yonezawa eds., *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, Cambridge, MA, 1994, pp. 151-171.

[Nord92] Robert Louis Nord. *Deriving and Manipulating Module Interfaces* (Ph.D. thesis). Carnegie Mellon University, CMU-CS-92-126, May 1992.

[Obraczka93] K. Obraczka, P. B. Danzig, and S-H Li. Internet Resource Discovery Services. *IEEE Computer*, September 1993, pp. 8-22.

[OMG91] Object Management Group. *Common Object Request Broker: Architecture and Specification.* OMG Document Number 91.12.1, 1991.

[OSF90] Open Software Foundation. *OSF/Motif Programmer's Reference. Revision 1.0.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Parnas72] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems Into Modules. *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp. 1053-1058.

[Perry87] Dewayne E. Perry. Software Interconnection Models. In *Proceedings, 9th International Conference on Software Engineering*, 1987, pp. 61-69.

[Perry89] Dewayne E. Perry. The Inscape Environment. In *Proceedings, 11th Int'l Conference on Software Engineering*, 1989, pp.2-12.

[Perry92] Dewayne Perry and Alexander Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 4, October 1992, pp. 40-52.

[Pree95] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, ACM Press, 1995.

[Purtilo94] James M. Purtilo. The POLYLITH Software Bus. *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 1, January 1994, pp. 151-174.

[Raynal86] M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press, Cambridge, MA, 1986.

[Rice94] M. D. Rice and S. B. Seidman. A Formal Model for Module Interconnection Languages. *IEEE Transactions on Software Engineering*, Vol. 20, No. 1, January 1994, pp. 88-101

[Rich90] Charles Rich and Richard C. Waters. *The Programmers Apprentice.* ACM Press Frontier Series, 1990.

[Scheifler88] R. W. Scheifler, J. Gettys, and R. Newman. *X Window System. C Library and Protocol Reference.* Digital Press. 1988.

[Sendoukas94] Hippocrates Sendoukas. *Usage Notes for DVIWIN 2.9.* December 1994.

[Shapeware94] *Using Visio 3.0.* Shapeware Corporation, Seattle, WA. 1994.

[Shaw89] Mary Shaw. Larger scale systems require larger scale abstractions. In *Proceedings, Fifth International Workshop on Software Specification and Design*, IEEE Computer Society, Software Engineering Notes 14, 3, May 1989, pp. 143-146.

[Shaw94a] Mary Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. Carnegie Mellon University, Technical Report CMU-CS-94-107. January 1994.

[Shaw94b] Mary Shaw and David Garlan. Characteristics of Higher-level Languages for Software Architecture. Technical Report CMU-CS-94-210. Also appears as CMU/SEI-94-TR-23, ESC-TR-94-023.

[Shaw95] Mary Shaw et. al. Abstractions for Software Architecture and Tools to Support Them. Paper in Progress, Carnegie Mellon University. Version of March 8, 1995.

[Stovsky88] M. P. Stovsky and B. W. Weide. Building Interprocess Communication Models Using STILE. *In Proceedings, 21st Annual Hawaii Int. Conf. on System Sciences*, 1988, Vol. 2, pp. 639-647.

[Sugimoto92] S. Sugimoto, T. Sakaguchi and Koichi Tabata. Layered Architecture of Multiple Programming Language System for Multiparadigm Programming. In *Proceedings, 1992 Int'l Conference on Computer Languages*, pp. 190-199.

[Tichy79] Walter F. Tichy. Software Development Control Based on Module Interconnection. *In Proceedings, 4th Int'l Conference on Software Engineering*, 1979, pp. 29-41.

[Wileden91] J. Wileden, A. Wolf, W. Rosenblatt and P. Tarr. Specification-Level Interoperability. *Communications of the ACM*, Vol. 34, No. 5, May 1991, pp. 72-87.

[Wyner95a] G. Wyner and G. Zlotkin. Resource, Use and Coordination. Presentation given at the MIT Center for Coordination Science, April 28, 1995.

[Wyner95b] G. Wyner and J. Lee. Applying Specialization to Process Models. In *Proceedings, Conference on Organizational Computing Systems*, August 13-16, 1995, Milpitas, CA, pp. 290-301.

[Yau83] S. S. Yau and M. U. Caglayan. Distributed Software System Design Representation Using Modified Petri Nets. *IEEE Transactions in Software Engineering*, Vol. SE-9, No. 6, November 1983, pp. 733-745.

[Yonezawa86] A. Yonezawa, J-P. Briot, and E. Shibayama. Object-Oriented Concurrent Programming in ABCL/1. *SIGPLAN Notices*, Vol. 21, No. 11, pp. 258-268.

[Zave89] Pamela Zave. A Compositional Approach to Multiparadigm Programming. *IEEE Software*, September 1989, pp. 15-25.