A Coordination Perspective on Software System Design

Chrysanthos Dellarocas
Sloan School of Management
Massachusetts Institute of Technology
Room E53-315, Cambridge, MA 02139, USA
Tel: +1 617 258-8115
dell@mit.edu

ABSTRACT
In large software systems the identification and proper management of interdependencies among the pieces of a system becomes a central concern. Nevertheless, most traditional programming languages and tools focus on representing components, leaving the description and management of interdependencies among components implicit, or distributed among the components. This paper identifies a number of problems with this approach and proposes a new perspective for designing software, which elevates the representation and management of component interdependencies to a distinct design problem. A core element of the perspective is the development of *design handbooks*, which catalogue the most common kinds of interconnection relationships encountered in software systems, as well as sets of alternative coordination protocols for managing them. The paper presents SYNTHESIS, a software development environment based on this perspective. A number of experiments performed using SYNTHESIS as well as some directions for future research are discussed.

Keywords
component-based software, software interconnection, software architecture, software reuse, coordination protocol

INTRODUCTION

In large software systems the identification and proper management of interconnection relationships and constraints among various pieces of a system has become responsible for an increasingly important part of the development effort. In many cases, the design, testing, and maintenance of protocols for managing communication, resource sharing, synchronization and other such interconnection relationships takes far more time and effort than the development of the core functional pieces of an application. In the rest of the paper, we will use the term *dependencies* to refer to interconnection relationships and constraints among components of a software system.

As design moves closer to implementation, current design and programming tools increasingly focus on components, leaving the description of interdependencies among components implicit, and the implementation of protocols for managing them fragmented and distributed in various parts of the system. 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 (procedures, packages, objects, clusters etc.), they are all essentially abstractions for components.

Most programming languages directly support a small set of primitive interconnection mechanisms, such as procedure calls, method invocation, shared variables, etc. Such mechanisms are not sufficient for managing more complex dependencies that are commonplace in today's software systems. Complex dependencies require the introduction of more complex managing protocols, typically comprising several lines of code (for example, the existence of a shared resource dependency might require a distributed mutual exclusion protocol). By failing to support separate abstractions for representing such complex protocols, current programming languages force programmers to distribute and embed them inside the interacting components [20] (Figure 1). Furthermore, the lack of means for representing dependencies and protocols for managing them has resulted in a corresponding lack of theories and systematic taxonomies of interconnection relationships and ways of managing them.

This expressive shortcoming of current languages and tools is directly connected to a number of practical problems in software design:

Accepted at the International Conference on Software Engineering and Knowledge Engineering (SEKE97), Madrid, Spain, June18-20, 1997.

- *Discontinuity between architectural and implementation models.* There is currently a gap between architectural representations of software systems (sets of activities explicitly connected through rich vocabularies of informal relationships) and implementation-level descriptions of the same systems (sets of modules implicitly connected through defines/uses relationships).

- *Difficulties in application maintenance*. By not providing abstractions for localizing information about dependencies, current languages force programmers to distribute the protocols for managing them in a number of different places inside a program. Therefore, in order to understand or modify a protocol, programmers have to look at many places in the program.

- *Difficulties in component reuse*. Components written in today's programming languages inevitably contain some fragments of coordination protocols from their original development environments. Such fragments act as undocumented assumptions about the structure of the application where such components will be used. When attempting to reuse such a component in a new environment, such assumptions might not match the interdependency patterns of the target application. In order to ensure interoperability, the original assumptions then 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 introduction of additional code around the component.

The preceding discussion forms the motivation for the rest of the paper. As a response to the problems identified in this section, we introduce the principles of our coordination perspective on software system design. Following that, we describe SYNTHESIS, a prototype software development environment which embodies the principles of the coordination perspective. We present our experience with using SYNTHESIS to facilitate software reuse. We discuss related work, describe some future directions of the project, and conclude the paper with a summary of our findings.

A COORDINATION PERSPECTIVE ON SOFTWARE SYSTEM DESIGN

The practical problems discussed in the previous section are rooted in the failure of most current programming languages and methodologies to recognize the identification and management of dependencies among software components as a design problem in its own right. This shortcoming translates to inadequate support for making interconnection assumptions embedded inside components more explicit, inability to localize information about interconnection protocols, and a lack of theories and systematic taxonomies of software interconnection relationships and ways of managing them.

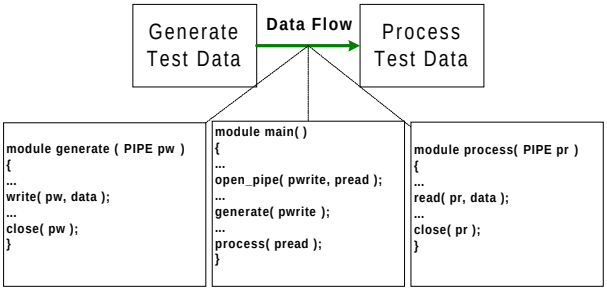

Figure 1: Implementation languages often force the distribution of coordination protocols among several code modules. In this example, the implementation code of a pipe protocol for managing a single data flow dependency has been distributed among three code modules.

As a response to this situation, this paper proposes a new perspective for representing and implementing software systems. Unlike current practice, this perspective emphasizes the explicit representation and management of dependencies among software activities as distinct entities.

The perspective is based on the ideas of coordination theory [12] and the Process Handbook project [3,11]. In accordance with Malone and Crowston [12], we define *coordination* as the act of managing dependencies among activities. In this case, the activities we are concerned with are software components. We will also use the term *coordination process* or *protocol* to describe the additional code introduced into a software system as a result of managing some dependency. Using these definitions, the principles of our coordination perspective on software system design can be stated as follows:

- *Explicitly represent software dependencies*. Software systems should be described using representations that clearly separate the core functional pieces of an application from their interdependencies, providing distinct abstractions for each.

- *Build design handbooks of component integration.* The field knowledge on component integration should be organized in systematic taxonomies that provide guidance to designers and facilitate the generation of new knowledge. Such taxonomies will catalogue the most common kinds of interconnection relationships encountered in practice. For each relationship, they will contain sets of alternative coordination protocols for managing it. In that way, they can form the basis for *design handbooks of component integration*, similar to the well-established handbooks that assist design in more mature engineering disciplines.

The long-term goal of this research is to develop concrete software development tools and methodologies based on

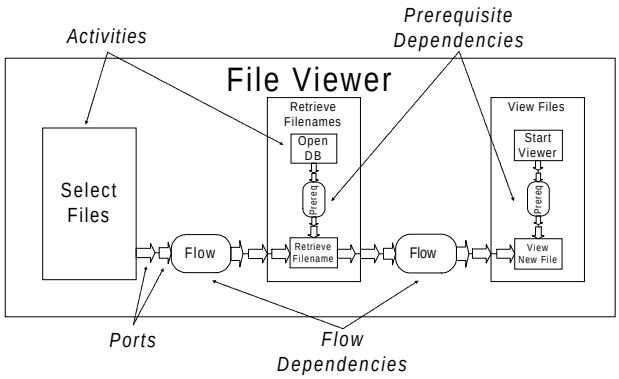

Figure 2: Representation of a simple file viewer application using SYNOPSIS.

the principles stated above, and to demonstrate that such methodologies provide practical benefits in the initial development, maintenance, and reuse of software systems. A crucial part of our efforts revolves around the definition of useful taxonomies of dependencies and coordination protocols and the organization of that knowledge in on-line repositories that can then assist or even automate the integration of the different parts of a system into a coherent whole. The rest of the paper describes our accomplishments and experience so far, as well as our plans for the future.

THE SYNTHESIS APPLICATION DEVELOPMENT ENVIRONMENT

The coordination perspective on software design introduced in the previous section has been reduced to practice by building SYNTHESIS, a prototype application development environment based on its principles. This section is devoted to a brief description of the SYNTHESIS system. A more detailed description can be found in [4].

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 developed using different environments (Intellicorp's Kappa-PC, Microsoft's Visual Basic, and Shapeware's Visio).

SYNTHESIS consists of three elements:

- SYNOPSIS, a software architecture description language
- an on-line "design handbook" of dependencies and associated coordination protocols
- a design assistant which generates executable applications by successive specializations of their SYNOPSIS description

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*. 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. For example, Figure 2 shows the SYNOPSIS description of a simple software system.

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).

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. Ports usually represent interfaces through which resources are produced and consumed by various activities. 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.

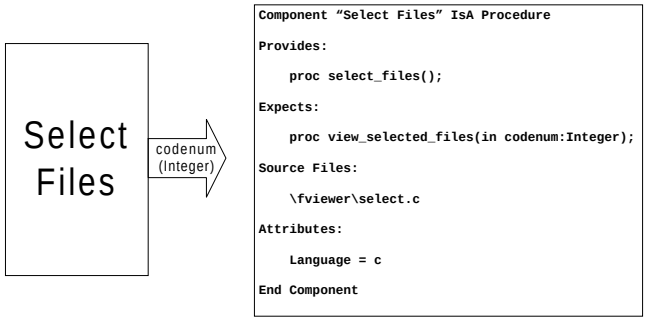

Figure 3: Example of an atomic activity and its associated code-level component description.

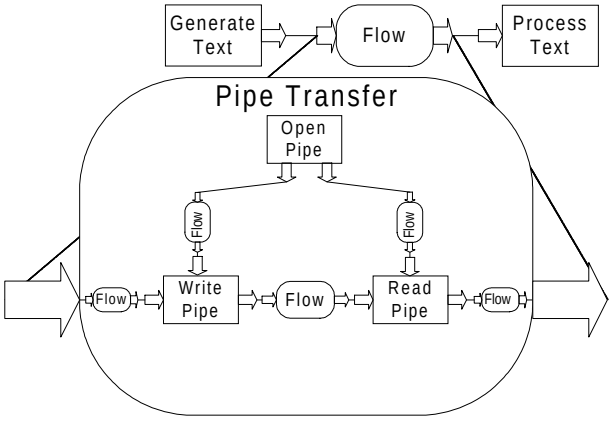

Figure 4: SYNOPSIS representation of a data flow dependency and its associated pipe transfer coordination protocol.

- An (optional) *component description*. Component descriptions associate 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. SYNOPSIS provides a special notation for describing the properties of software components associated with executable activities. Such properties include the component kind, the provided and expected interfaces of the component, source and object files needed by the component, etc. (Figure 3).

Depending on the values of the above two attributes, activities are distinguished as follows:

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

Dependencies

Dependencies describe interconnection relationships and constraints among activities. 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 as the composite dependency.
- An (optional) *coordination protocol*. Coordination protocols are patterns of simpler dependencies and activities that describe a mechanism for managing the relationship or constraint implied by the dependency (Figure 4).
- 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.

In a manner similar to activities, dependencies are distinguished into atomic or composite, executable or generic.

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 entities 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 [3, 11].

The mechanism of entity specialization enables the creation of specialization hierarchies for activities, dependencies, ports, and coordination protocols. 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 5).

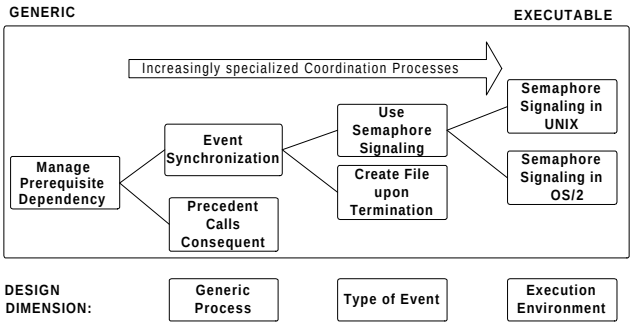

Figure 5: A hierarchy of prerequisite dependencies with increasingly specialized associated coordination protocols.

A design handbook of software component interconnection

SYNTHESIS contains our initial version of a handbook of dependencies and coordination protocols commonly encountered in software systems. The prototype handbook is stored on-line as a hierarchy of increasingly specialized SYNOPSIS dependency entities. An entity browser interface (similar to class browsers of object-oriented systems) is available for navigation and selection of handbook entries.

An important decision in making a taxonomy of software interconnection is the choice of the generic dependency types. If we are to treat software interconnection as an orthogonal problem to that of designing the core functional components of an application, dependencies among components should represent relationships which are also orthogonal to the functional domain of an application. Fortunately, this requirement is consistent with the nature of most interconnection problems: Whether our application is controlling inventory or driving a nuclear submarine, most problems related to connecting its components together are related to a relatively narrow set of concepts, such as resource flows, resource sharing, and timing dependencies. The design of associated coordination protocols involves a similarly narrow set of mechanisms such as shared events, invocation mechanisms, and communication protocols.

The prototype handbook is based on the simple assumption that software component interdependencies are explicitly or implicitly related to patterns of resource production and usage. Beginning from this assumption, we have defined a number of useful dependency families in a way independent of the application context where they might be used. Dependency families represented in the handbook include:

- *Flow dependencies*. Flow dependencies represent relationships between producers and consumers of resources. They are specialized according to the kind of resource, the number of producers, the number of consumers, etc. Coordination protocols for managing flows decompose into protocols which ensure accessibility of the resource by the consumers (usually by physically transporting it across a communication medium), usability of the resource (usually by performing appropriate data format conversions), as well as synchronization between producers and consumers.
- *Sharing dependencies*. They encode relationships among consumers who use the same resource. Sharing dependencies are specialized according to the sharing properties of the resource in use (divisibility, consumability, concurrency). Coordination protocols for sharing dependencies ensure proper enforcement of the sharing properties, usually by dividing a resource among competing users, or by enforcing mutual exclusion protocols.
- *Timing dependencies*. Timing dependencies express constraints on the relative flow of control among a set of activities. Examples include *prerequisite dependencies* and *mutual exclusion dependencies*. Timing dependencies are used to specify application-specific cooperation patterns among activities which share the same resources. They are also used in the decomposition of coordination protocols for flow and sharing dependencies.

A detailed description of the contents of the prototype handbook can be found in [4].

SYNTHESIS Design Assistant

The design methodology supported by SYNTHESIS uses SYNOPSIS descriptions in order to both specify and implement software systems. SYNTHESIS supports a process for generating executable systems by successive specialization of their SYNOPSIS descriptions. The process is summarized in Figure 6. As can be seen, the existence of on-line design handbooks of activities and dependencies can assist, and often automate, parts of the process.

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. They can optionally input an *evaluation function* that helps the system perform an automatic ranking and selection of compatible candidates. Furthermore, successive transformations of the original application diagram (stored

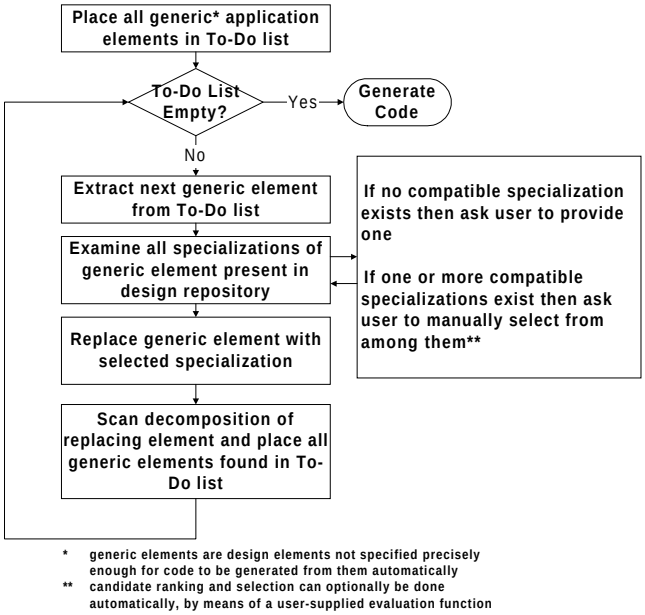

Figure 6: Sketch of an algorithm used by SYNTHESIS to generate executable applications by successive specializations of their SYNOPSIS descriptions.

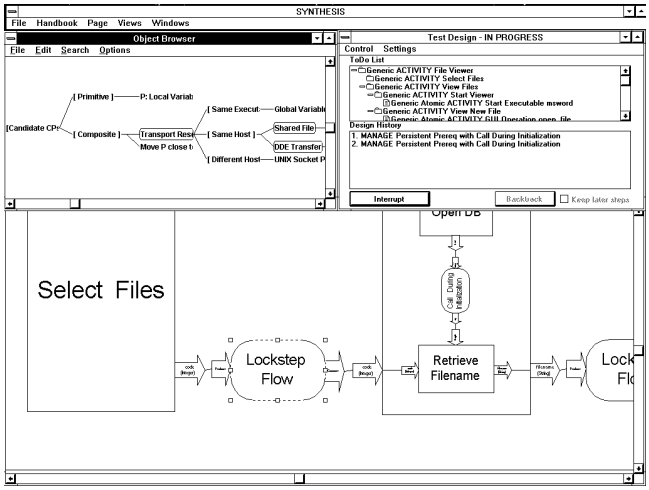

Figure 7: Configuration of SYNTHESIS windows during design mode

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.

Figure 7 shows the layout of SYNTHESIS windows during design. 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, the *design manager* window (in the upper right part of the screen), summarizes the status of the design process and allows users to control its parameters.

USING SYNTHESIS TO FACILITATE COMPONENT-BASED SOFTWARE DEVELOPMENT

The first domain where we tested the practical advantages of our coordination perspective on software system design is the development of new applications by integrating existing software components. This section describes the experience gained by using SYNTHESIS to facilitate the reuse of existing software components in new applications.

One of the important practical difficulties of building new systems by reusing existing parts lies in the amount of effort required in order to bridge mismatches among components. In most cases, some additional "glue code" needs to be added to integrate all the independently written software pieces into a coherent application [5]. An application development tool based on the principles of our coordination perspective on software design has the potential of helping alleviate the difficulty of component integration by separating the representation of activities and dependencies, localizing information related to coordination protocols, and providing frameworks of common dependencies and coordination protocols.

To test this claim, we have used SYNTHESIS in order to build a set of test applications by reusing independently written pieces of software. 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 experiments are described in full detail in [4]. Table 1 provides a brief summary.

Experiment 1 consisted in building a simple File Viewer application by combining a commercial text editor and pieces of code written in C and Visual Basic. It 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 investigated 9 different ways of building an indexing system by various combinations of server and UNIX filter components (Table 2). It provided positive evidence for the ability of the system to resolve *architectural mismatches*, that is, different assumptions about the structure of the application in which they will be used. It 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.

Experiment 3 combined the standard components of the TEX document typesetting system in a WYSIWYG application. It tested the power of SYNOPSIS and our proposed vocabulary of dependencies in expressing non-trivial application architectures.

Finally, Experiment 4 attempted to build a collaborative editor by extending the functionality of an existing text editor. It investigated the usefulness of the system in assisting the rapid development of applications for multiple platforms. It demonstrated that different implementations of the same application, suitable for different execution

Experiment	*Description*	*Components*	*Results*
File Viewer	A simple system which retrieves and displays the contents of user-selected files.	User interface component written in C; filename retrieval component written in Visual Basic; file display component implemented using commercial text editor.	SYNTHESIS integrated components suggesting two alternative organizations (client/server, implicit invocation); all necessary coordination code was automatically generated in both cases.
Key Word in Context	A system that produces a listing of all circular shifts of all input lines in alphabetical order [18].	Two alternative implementations for each component (both written in C): as a server and as a UNIX filter.	3 different combinations of filter and server implementations were each integrated in 3 different organizations (see Table 2). SYNTHESIS generated most coordination code; users had to manually write 16 lines of code in 2 cases
Interactive TEX	A system that integrates the standard components of the TEX document typesetting system in a WYSIWYG ensemble.	Standard executable components of TEX system.	Target application was completely described in SYNOPSIS. SYNTHESIS was able to generate coordination code automatically.
Collaborative Editor	A system which extends the functionality of existing single user editors with group editing capabilities [7].	Micro-Emacs [10] source code was used to implement single-user editor.	Same system description was specialized in two different ways to generate micro-Emacs-based group editors for Windows and UNIX.

Table 1: Summary of experiments of using SYNTHESIS to facilitate the integration of existing software components in new applications.

environments, can be generated from the same, partially specialized SYNOPSIS system description, by selecting different coordination processes for managing dependencies.

Overall, our experiments 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.

RELATED WORK

The ideas expressed in this work are most closely related to research in coordination theory and architecture description languages. Recent efforts to build open software architectures are an interesting, but contrasting, approach for achieving many of the goals of our coordination perspective on software system design. This section briefly discusses all three research areas.

Coordination Theory

Coordination theory [12] 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,

	Components	*Architecture*	*Auto lines**	*Manual lines†*
1	Filters	Pipes	34	0
2	Filters	Main Prog/Subroutine	30	0
3	Filters	Implicit Invocation	150	0
4	Servers	Pipes	78	16
5	Servers	Main Prog/Subroutine	35	0
6	Servers	Implicit Invocation	95	0
7	Mixed	Pipes	66	0
8	Mixed	Main Prog/Subroutine	56	0
9	Mixed	Implicit Invocation	131	16

* Lines of coordination code automatically generated by Synthesis
† Lines of coordination code manually added by user

Table 2: Summary of the Key Word in Context experiments

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 protocols that can be used to manage them.

The present work can be viewed as an application and extension of 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 protocols. The project grew out of the Process Handbook project [3, 11] 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 protocols for the specialized domain of software systems.

Architecture Description Languages

Several Architecture Description Languages (ADLs) provide support for representing software systems in terms of their components and their interconnections [8]. Different languages define interconnections in different ways. For example, Rapide [21] connections are mappings from services required by one component to services provided by another component. Unicon [22] connectors define protocols that are inserted into the system in order to integrate a set of components. In that sense they are similar to the coordination protocols that manage dependencies in SYNTHESIS. Like Unicon, SYNTHESIS views dependencies as relationships among components which might require the introduction of additional coordination code in order to be properly managed. Unlike Unicon, however, SYNTHESIS dependencies are specifications which can then be managed (i.e. implemented) in a number of different ways. The set of dependency types is not fixed. Coordination theory is a framework that assists the discovery of additional dependency types and coordination protocols. Finally, apart from simply supporting dependency representations, the work reported in this paper proposes the development of taxonomies of abstract dependency relationships and coordination protocols for managing them as a key element in facilitating component-based software development.

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. Such approaches are based on standardizing some part of the glue required to compose components. The most notable efforts in that direction include object-oriented architecture standards, such as CORBA [16], Microsoft's OLE [15], and Apple's Open Scripting Architecture [1], and application frameworks such as X-Windows/Motif [17, 19] and Microsoft Visual Basic [14].

Open software architectures and our coordination perspective were both motivated by the complexity of managing component interdependencies. However, the two approaches represent very different philosophies. Open architectures take the stance that designers should not have to deal with software dependencies. In essence they are "hiding interconnection protocols under the carpet" by limiting the kinds of allowed relationships and by providing a standardized infrastructure for managing them. Our coordination perspective, in contrast, is based on the belief that the identification and management of software dependencies should be elevated to a design problem in its own right. Therefore, dependencies should not only be explicitly represented as distinct entities, but furthermore, when deciding on a managing protocol, the full range of possibilities should be considered with the help of design handbooks.

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. Third, the standardized interaction protocols might not be optimal for all applications.

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 protocol exists in its repository, SYNTHESIS will be able to interconnect them. Furthermore, SYNTHESIS is able to suggest several alternative ways of managing an interconnection relationship and thus possibly generate more efficient implementations. Finally, open software architecture protocols can be incorporated into SYNTHESIS repositories as special cases of coordination protocols.

FUTURE RESEARCH

The long-term goal of this research is to demonstrate the practical usefulness of a coordination perspective on software system design and to develop superior software development methodologies based on its principles. To that end, this section describes some immediate directions for future research suggested by our experience so far.

- *Classify composite dependency patterns*. Our current taxonomy includes relatively low-level dependency types, such as flows and prerequisites. In a sense, our taxonomy defines a vocabulary of software interconnection relationships. A particularly promising path of research seems to be the classification of more complex dependency types as patterns of more elementary dependencies.
- *Develop coordination process design rules.* It will be interesting to develop design rules that help automate the 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.
- *Develop guidelines for better reusable components*. The idea of separating the design of component functionality from the design of interconnection protocols has interesting implications about the way reusable components should be designed in the future. At best, components should contain minimal assumptions about their interconnection patterns with other components embedded in them. More research is needed to translate this abstract requirement to concrete design guidelines.

CONCLUSIONS

This work was motivated by the increasing variety and complexity of interdependencies among components of large software systems. It has observed that most current programming languages and tools do not provide adequate support for identifying and representing such dependencies, while the knowledge of managing them has not yet been systematically codified.

The initial results of this research provide positive evidence for supporting the claim that software component integration can usefully be treated as a design problem in its own right, orthogonal to the specification and implementation of the core functional pieces of an application.

More specifically, software interconnection dependencies and coordination protocols for managing them can be usefully represented as independent entities, separate from the interdependent components.

Furthermore, common dependency types and ways of managing them can be systematically organized in a design handbook. Such a handbook, organized as an on-line repository, can assist, or even automate the process of transforming architectural descriptions of systems into executable implementations by successive specializations.

Our experience with building SYNTHESIS, a prototype application development environment based on these principles and using it as a tool for facilitating the reuse of existing components in new applications has demonstrated both the feasibility and the practical usefulness of this approach. With our future research we plan to expand and refine the contents of our design handbook of dependencies and coordination processes as well as investigate the usefulness of our approach in larger-scale software systems.

ACKNOWLEDGMENTS

The author would like to acknowledge the support and invaluable help of Thomas W. Malone in supervising the doctoral thesis which formed the basis for the work reported in this paper.

REFERENCES

1. Apple Computer. *Inside Macintosh Volume 7: Interapplication Communication*. Addison-Wesley, 1993.
2. T. J. Biggerstaff and A. J. Perlis. *Software Reusability*. Volumes 1 and 2, ACM Press/Addison Wesley, 1989.
3. 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.

4. Chrysanthos Dellarocas. *A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating Software Components* (Ph.D. Thesis). MIT Center for Coordination Science Working Paper #193, February 1996.

5. 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.

6. 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.

7. 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.

8. 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.

9. Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, Vol. 24, No. 2, June 1992, pp. 131-183.

10. D. M. Lawrence and B. Straight. *MicroEmacs Full Screen Text Editor Reference Manual, version 3.10*, March 1989.

11. 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.

12. Thomas W. Malone and Kevin Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, Vol. 26, No. 1, March 1994, pp. 87-119.

13. 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.

14. *Language Reference, Microsoft Visual Basic Version 3.0*. Microsoft Corporation, Redmond, WA. 1993.

15. *OLE2 Programmers' Reference*, Vols. 1 and 2, Microsoft Press, Redmond, Wash., 1994.

16. Object Management Group. *Common Object Request Broker: Architecture and Specification*. OMG Document Number 91.12.1, 1991.

17. Open Software Foundation. *OSF/Motif Programmer's Reference. Revision 1.0*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

18. 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.

19. R. W. Scheifler, J. Gettys, and R. Newman. *X Window System. C Library and Protocol Reference*. Digital Press. 1988.

20. 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.

21. David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, Vol. 21, No. 9, September 1995, pp. 717-734.

22. Mary Shaw, Robert DeLine, and Daniel Klein. Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions of Software Engineering 21, 4, April 1995, pp. 314-335.