The SYNTHESIS Environment for Component-Based Software Development

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

Abstract:

Component-based software development places an emphasis on identifying and managing interdependencies among preexisting pieces of software in order to integrate them into new systems. Traditional software development methodologies, on the other hand, place an emphasis on representing components, leaving the description and management of component interdependencies implicit, or distributed among the components. To support component-based software development, we need new methodologies and tools which elevate the representation and management of software component interdependencies to a distinct design problem, orthogonal to the specification and implementation of the core functional pieces of an application. A core element of such methodologies will be a "design handbook" of software component interconnection, which catalogues common software interconnection dependencies and sets of alternative protocols for managing them. SYNTHESIS, a component-based software development environment based on this perspective, has been developed and successfully used to minimize the manual effort required to integrate independently developed components into new applications.

1. Introduction

During the past few years a range of technical, economic, and social factors have come together to encourage a new way of software engineering that is often referred to as component-based software engineering. This approach bases system development on the definition of software architectures that capture the needs of a given organization and on the selection and integration of components that implement the pieces of these architectures. The big promise of component-based software engineering lies in the possibility to reuse independently developed, off-the-shelf components in order to build new applications more rapidly, economically, and reliably than with traditional approaches.

Despite the significant economic potential and substantial research effort that has been put into component-based software engineering, so far this new paradigm of software development has failed to gain a significant presence in large-scale, commercial development projects [2, 6]. Part of the reason is related to the difficulty of locating appropriate components and the legal issues surrounding their reuse. But even when such issues have been resolved, the lack of software development methodologies specifically designed to support component-based development is discouraging many software engineers from adopting it.

This paper argues that traditional software development methodologies are not well suited to the requirements of component-based software development. It identifies the treatment of interdependencies among software components as one fundamental area where component-based development poses new requirements, not met by traditional methodologies and tools. It proposes a new perspective for developing software systems which treats the interconnection of components in a software system as a separate design problem, entitled to its own representations and design frameworks. It introduces SYNTHESIS, a software development environment based on our perspective and reports on experience gained by using the system to develop new applications from sets of existing components. Finally, it discusses related work and presents some directions for future research.

2. Requirements for Component-Based Software Development

In this section we identify a fundamental difference between traditional and component-based software development. We argue that this difference merits the

Presented at the 8th International Workshop on Software Technology and Engineering Practice (STEP'97), London, UK, July 14-18, 1997

development of new methodologies specifically tailored to the needs of the new paradigm. Finally, we introduce a perspective on which such methodologies can be based.

Both traditional and component-based software development methodologies need to specify the requirements and overall architecture of the target software system. However, from then on the two paradigms of software development focus on a different set of design activities.

The objective of traditional software development methodologies is to facilitate the *creation* of one or more implementation-level modules which, together, implement the functionality and data of a software system. Despite the diversity of the various methodologies in use today, they are all essentially providing models and techniques that help answer the following questions:

- How can we best divide the required functionality and data of a system into a set of components.
- How can we best encode the required interactions among components of a system into component interfaces.

Traditional methodologies focus on defining components, leaving the definition of interdependencies among components implicit, and the implementation of protocols for managing them fragmented and distributed among the interacting components. At the implementation level, software systems are sets of 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. 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 [14]. 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 dependencies and ways of managing them.

In component-based development, the components are usually pre-existing and fixed, or customizable in a limited way. The design focus then lies on *integrating* existing components to form new systems. The essential design questions that methodologies must help answer become the following:

- How can we best select existing components to implement a system's functional pieces.
- How can we best manage the interdependencies and mismatches among the selected components and integrate them into a seamless system.

In this paper we will focus on the second question. Traditional methodologies do not provide much help with answering this question because they do not recognize interdependencies as a distinct design entity, nor do they provide any systematic guidance for designing coordination protocols for managing such interdependencies. As a result, most component integration projects today are carried out in an ad-hoc manner, resulting in frequent time and budget overruns and the general perception that component integration is more difficult than it should be [5].

As a response to the previous observations, this paper proposes a new perspective for specifying and implementing software systems. This perspective can form the basis for practical component-based software development methodologies. It is based on coordination theory [12] and applies concepts developed in the Process Handbook project [4, 11]. Unlike current practice, our perspective emphasizes the explicit representation and management of dependencies among software activities as distinct entities. The two main principles of our perspective 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.

3. The SYNTHESIS Application Development Environment

The coordination perspective on software design introduced in the previous section has been reduced to practice by building SYNTHESIS, an application development environment based on its principles. SYNTHESIS is particularly well suited for component-based software development. This section presents a brief introduction to the SYNTHESIS system. A detailed description can be found in [3].

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

3.1 SYNOPSIS: An Architecture Description Language

SYNOPSIS supports graphical descriptions of software application architectures at both the specification and the implementation level. The language provides separate language entities for representing software *activities* and *dependencies*. It also supports the mechanism of *entity specialization*. Specialization allows new entities (activities and dependencies) 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 represent interfaces through which resources are produced and consumed by various activities.

An activity can optionally decompose into patterns of simpler activities and dependencies which implement the functionality intended by the composite activity. An activity can have an optional association with a code-level component which implements its intended functionality. Examples of code-level components include source code modules, executable programs, network services, etc. SYNOPSIS provides a special notation for describing the properties of software components associated with activities.

Activities are distinguished into generic and executable. *Executable* activities are activities which have either a direct association to a code-level component, or a decomposition whose members are all executable. *Generic* activities are activities that do not have a direct association to a code-level component and/or have a decomposition where at least one member is not executable.

Dependencies

Dependencies describe interconnection relationships and constraints among activities. Like activities they can optionally decompose into patterns of simpler dependencies. They can have optional associations with *coordination protocols*. Coordination protocols are activities that introduce the additional code required in order to manage their associated dependency.

Like activities, dependencies are also distinguished into generic and executable. *Executable (or managed)* dependencies are dependencies which have either a direct association to an executable coordination protocol, or a decomposition whose members are all executable. *Generic (or unmanaged)* dependencies are dependencies that do not have a direct association to an executable coordination protocol and/or have a decomposition where at least one member is not executable.

Using the above definitions, SYNOPSIS can be used both to describe system specifications (sets of generic activities and dependencies) as well as system implementations (sets of executable activities and dependencies). Furthermore, implementations can be derived from specifications by successive specializations of generic elements into executable. Figure 1 shows an example of a software application specification and implementation in SYNOPSIS.

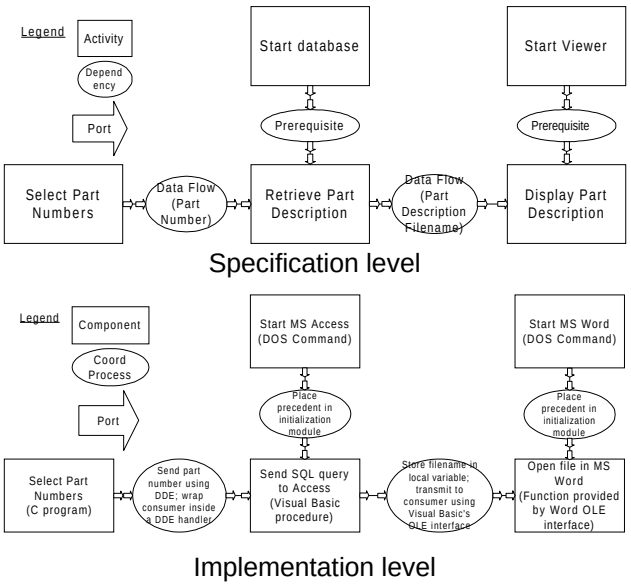

Figure 1: Representation of a simple file viewer application using SYNOPSIS.

3.2 A Design Handbook of Software Component Integration

The ability to represent dependencies and coordination protocols as distinct entities from the components they interconnect allows the construction of taxonomies that systematize the field knowledge in integrating software components, and provide guidance for solving such problems in a routine manner. Such taxonomies should contain:

- a catalog of the most common kinds of interconnection dependencies encountered in software systems
- for each kind of dependency, a catalog of sets of alternative coordination protocols for managing it

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.

After making a survey of existing systems, and building on earlier results of coordination theory [12], we have based the taxonomy of dependencies presented in this paper on the assumption that component interdependencies are explicitly or implicitly related to patterns of resource production and usage. In other words, activities need to interconnect with other activities, either because they use resources produced by other activities, or because they share resources with other activities.

Based on this assumption, the most generic dependency families in our taxonomy 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 or producers who produce for the same consumers. 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 our taxonomy of dependencies and coordination processes can be found in [3].

Our SYNTHESIS prototype contains an on-line version of our taxonomy of dependencies and coordination processes. The design spaces of our taxonomy have been

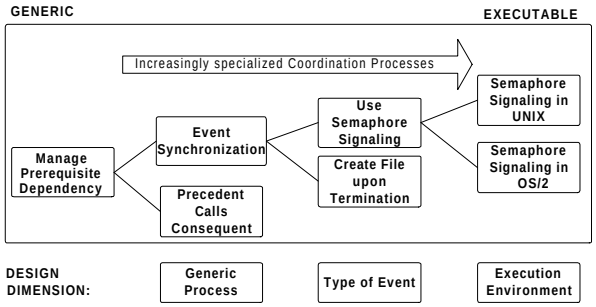

Figure 2: A hierarchy of increasingly specialized coordination protocols for managing prerequisite dependencies.

implemented by hierarchies of increasingly specialized SYNOPSIS entities. For example, Figure 2 shows a partial hierarchy of increasingly specialized processes for managing prerequisite dependencies. Each process contained in the handbook contains attributes that enable the system to automatically determine whether it is a compatible candidate for managing a dependency between a given set of components.

3.3 A Design Process for Generating Executable Applications

SYNTHESIS supports a process for generating executable systems by successive specialization of their SYNOPSIS descriptions. The process can be summarized as follows:

1. Users describe their application using SYNOPSIS, as a pattern of activities connected through dependencies.
2. The design assistant of SYNTHESIS scans the application description and iteratively does the following for each generic (i.e. not executable) application element:
   a) It searches the on-line design handbook for compatible specializations.
   b) It selects one of the compatible specializations found, either automatically, or by asking the user. If no compatible specialization can be found, it asks the user to provide one.
   c) It replaces the generic application element with the selected specialization and recursively applies the same process to all elements in the decomposition of this element.
3. After all application elements have been replaced by executable specializations, the design assistant integrates them into a set of modules in one or more languages and generates an executable application out of the collection.

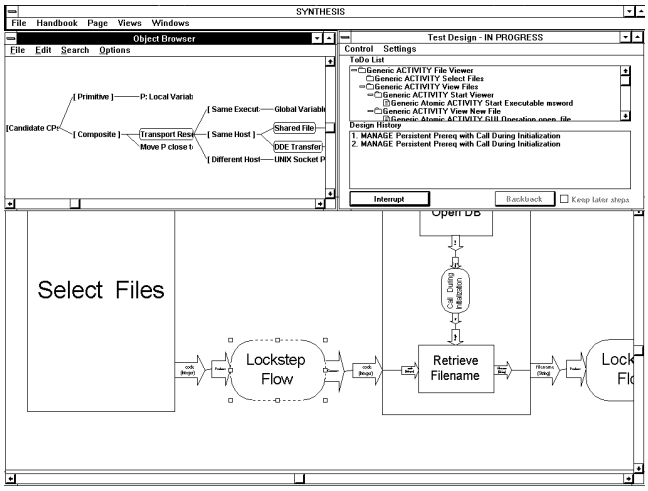

Figure 3: Configuration of SYNTHESIS windows during the design process

Figure 3 shows the configuration of SYNTHESIS windows during the design process.

The above design process minimizes the manual effort required to integrate software components into new systems. Users only need to participate in the specialization process by making the final selection when more than one compatible specializations have been found. In the rare cases when no compatible specialization can be found, users need to provide the code for such a specialization. Specializations thus provided become a permanent part of the repository.

4. Using SYNTHESIS to Facilitate Component-Based Software Development

4.1 Overview

We have tested the capabilities of SYNTHESIS by using it to build a set of applications by integrating independently written pieces of software. Each experiment consisted in:

- describing a test application as a SYNOPSIS diagram of activities and dependencies
- selecting a set of pre-existing components exhibiting various mismatches to implement activities

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.

- using the design process outlined above to semi-automatically manage dependencies and integrate the selected components into an executable system
- exploring alternative executable implementations based on the same set of components

The results of our experiments are summarized in Table 1. Overall, we used SYNTHESIS to build 4 test applications. Each application was integrated in at least two different ways. For example, for one application we built one implementation where components were organized around client/server interactions, and a second where the same components were organized around peer-to-peer interactions. This resulted in a total of 14 different implementations. SYNTHESIS was able to build all 14 implementations, typically generating between 30-200 lines of additional glue code in each case in order to manage interdependencies and integrate the components. In only 2 cases, users had to manually write 16 lines of code (each time), to implement two data conversion routines that were missing from the design handbook. Reference [3] contains a detailed description of our experiments.

4.2 Example: Building a Collaborative Editor by Integrating Existing Components

This section presents one of our experiments in more detail. It also serves as an example of how SYNTHESIS can form the basis for a methodology for component-based software development.

In our experiment, we used SYNOPSIS to create a collaborative editor architecture, loosely based on the ideas of Knister and Prakash [7]. Collaborative editors allow the joint concurrent editing of the same document by a group of people. We selected an existing single-user editor and mapped the activities of our architecture to source code modules of that system. Then, we used SYNOPSIS and its on-line design handbook of coordination protocols in order to manage the dependencies of our architecture and integrate the resulting set of components and coordination protocols into an executable collaborative editor application for Microsoft Windows. Finally, we reused the same collaborative editor architecture and used SYNTHESIS in order to generate an equivalent application for UNIX.

Our collaborative editor architecture implements a collaboration protocol loosely based on the one presented in [7]. 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.

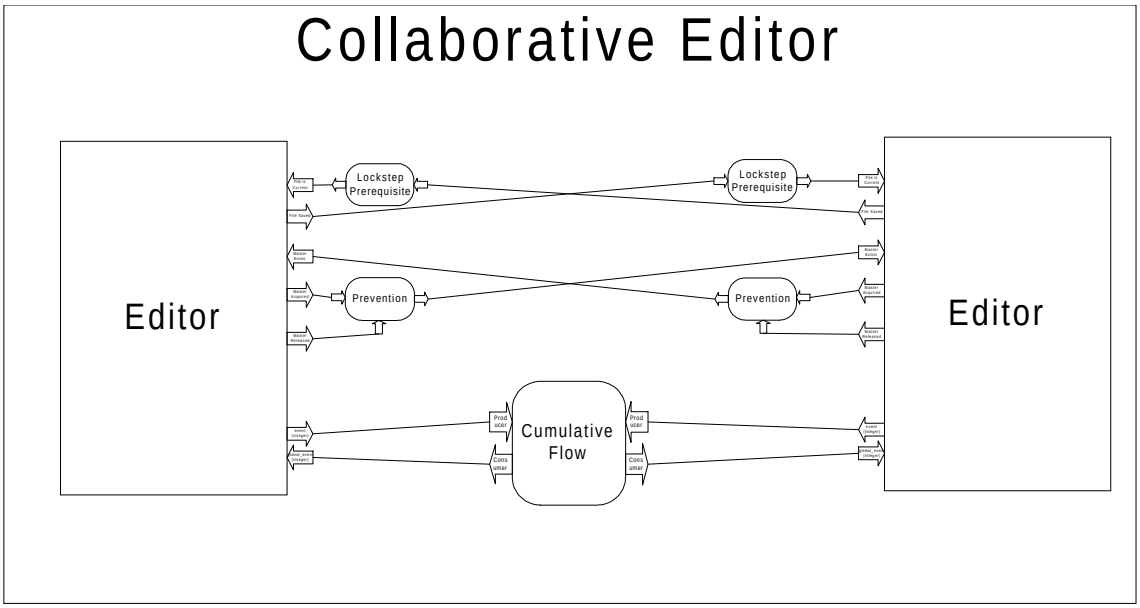

Figure 4 : Top level decomposition of collaborative editor architecture with two participants.

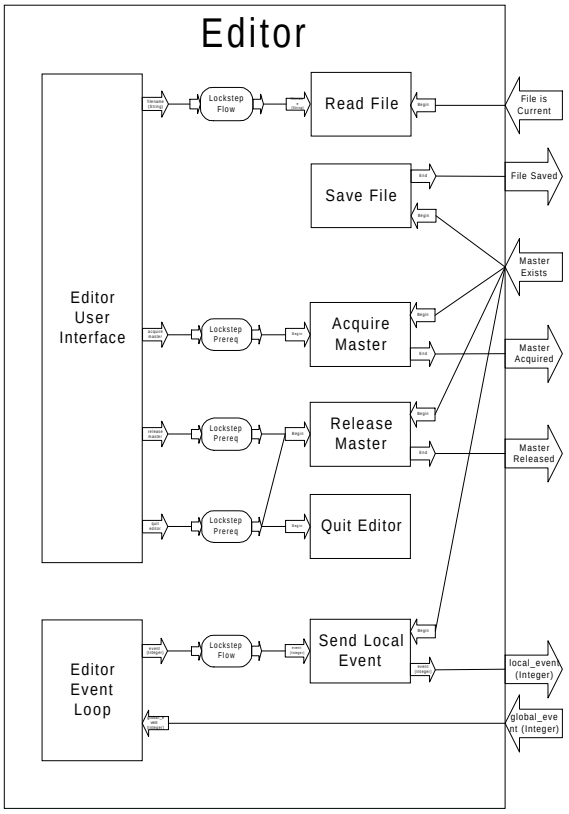

Figure 5: Decomposition of Editor activity.

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.

A SYNOPSIS architecture for describing a collaborative editor system which supports the collaboration protocol described above is given in Figure 4. The architecture interconnects a set of simpler Editor activities, corresponding to individual session participants. The decomposition of Editor activities is given in Figure 5.

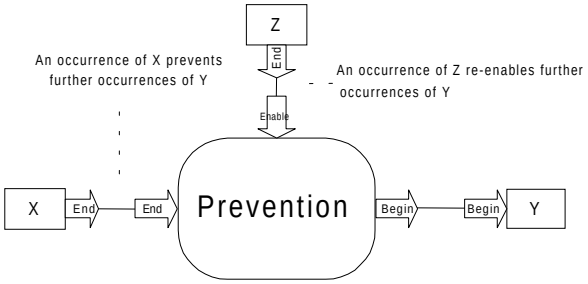

Figure 6: Prevention dependencies.

The operation of the overall system is based on a Flow dependency for broadcasting a participant's keystrokes to all participants, and a set of Prevention dependencies, each connecting a participant to all other participants (except itself). Prevention dependencies belong to the general family of timing dependencies of our taxonomy. They specify that the occurrence of an activity X (enabler) prevents further occurrences of an activity Y until a third activity Z (disabler) occurs, in which case occurrences of Y are re-enabled (Figure 6).

In our system, each Prevention 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 Prevention dependency prevents the execution of all Editor subactivities connected to the Master Exists port of all participants except the one who acquired master status. These subactivities include the ability to broadcast a participant's local keystrokes, as well as the ability to acquire and release master status. The resulting effect is that only the current master can make its keystrokes visible to everyone's local editor copy and only the master can release master status.

Each Editor activity can be 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. The activities depicted in Figure 5 must be mapped to source code modules of the single-user editor using the component description language of SYNOPSIS (see [3] for details).

In our experiment we used MicroEmacs [10] as our single-user editor component. 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.

Each version of MicroEmacs was "fitted" into our generic architecture with a minimal need for manual modifications, very similar to those described in [7]. Then SYNOPSIS was able to manage the same set of dependencies with coordination protocols specific for each execution environment in order to generate two alternative executable implementations of the same system (one for UNIX, one for Windows).

The experiment provided an excellent example of how component-based software development can facilitate the reuse of software architectures in order to facilitate the generation of applications for multiple platforms.

5. Related Work

5.1 The Process Handbook Project

The work reported in this paper grew out of the Process Handbook project at MIT's Center for Coordination Science [4, 11]. The Process Handbook project applies the ideas of coordination theory [12] 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 revolves around a software tool, called the "process handbook", which contains rich descriptions of how different organizations perform similar processes, including the relative advantages of the alternatives. SYNOPSIS has borrowed the ideas of separating activities from dependencies and the notion of entity specialization from the Process Handbook. SYNOPSIS has moved beyond the Process Handbook in refining the process representation so that it can describe software applications at a level precise enough for code generation to take place, and in defining a repository of dependencies and coordination protocols for the specialized domain of software systems.

5.2 Architecture Description Languages

Architecture Description Languages (ADLs) provide support for representing software systems in terms of their components and their interconnections [8, 15]. They often provide separate abstractions for representing components and their interconnections. SYNOPSIS shares many of the goals and principles of many recent ADLs, most notably UniCon [16]. 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). Furthermore, apart from introducing a new architectural language, this work proposes a more general perspective on designing systems which also

includes the development of design handbooks for activities and dependencies as well as a design process for generating executable systems by successive specializations of their architectural descriptions.

5.3 Component Frameworks

Component frameworks such as OLE, CORBA, OpenDoc, etc. [1] facilitate the interoperation of independently developed components by limiting the kinds of allowed interactions and by providing a standardized infrastructure for managing them.

Component frameworks and our coordination perspective represent two very different philosophies in component-based software development. Component frameworks, once again, place the emphasis on components. They provide a fixed infrastructure for managing component interdependencies (middleware) and require components to adhere to a specific set of standards in order to interoperate.

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. 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 interconnection protocols defined in specific component frameworks can be incorporated into SYNTHESIS repositories as one, out of many, alternative ways of managing the underlying dependency relationships.

6. Discussion and Future Work

Component-based software engineering builds new software systems by integrating existing components. Identifying and properly managing the interdependencies among components becomes a central concern in this new paradigm. Software engineering methodologies specifically geared towards component-based development need to elevate the representation and management of interdependencies among software components to a distinct design problem, entitled to its own abstractions and design taxonomies.

SYNTHESIS is one such methodology and toolset. It is based on an architecture description language that clearly separates software activities and dependencies, and on a design handbook that contains the most common types of dependencies encountered in software systems, as well as sets of alternative coordination processes for managing them.

The practical advantages of SYNTHESIS include:

- *Easier integration of code-level components.* SYNTHESIS can take advantage of its on-line design handbook (a systematic codification of field knowledge in component integration) in order to minimize the need for additional manually written code to bridge mismatches and manage dependencies among software components.
- *Support for rapid multi-platform development*. When applications are ported to a new execution environment, their abstract architecture (activities and dependencies) remains unaffected. The parts most likely to require modification are the coordination protocols that manage their dependencies. By expressing new applications as SYNOPSIS diagrams, dependencies can be managed using alternative coordination protocols in order to generate code for multiple platforms beginning from a single system description. The experiment discussed in Section 4.2 provided an example of how SYNTHESIS can be used to generate code for multiple platforms from a single SYNOPSIS description.
- *Superior insight into the range of alternative implementations*. The explicit representation of dependencies as distinct entities which are subsequently managed by consulting a design handbook, encourages designers to consider a wide range of alternative ways of implementing their systems. We are confident that this will result in a more systematic and rational way of organizing large-scale systems.
- *Easier application maintenance*. Designers often need to change the implementation of activities, in order to reflect changes in functional requirements or evolutions in component implementation. Applications will 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.

Our initial experience with SYNTHESIS has provided positive evidence to support our claims on small-scale systems. With our future research we plan to demonstrate the advantages of our approach as a basis for building and

maintaining large-scale software systems out of existing parts.

References

1. Richard M. Adler. Emerging Standards for Component Software. *IEEE Computer*, March 1995, pp. 68-77.
2. T. J. Biggerstaff and A. J. Perlis. *Software Reusability*. Volumes 1 and 2, ACM Press/Addison Wesley, 1989.
3. 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. Also available from http://ccs.mit.edu/ccswp193/main.html
4. 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.
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. 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.
14. 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.
15. 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.
16. 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.