Toward a Design Handbook for Integrating Software Components

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:

In component-based software development the identification and proper management of interconnections among the pieces of a system becomes a central concern. Nevertheless, today's programming languages and tools still place an emphasis on representing components, leaving the description and management of component interdependencies implicit, or distributed among the components. This paper proposes a new perspective for designing software which elevates the representation and management of software component interdependencies to a distinct design problem, orthogonal to that of representing and implementing the core functional pieces of an application. The perspective is based on a taxonomy of common software interconnection dependencies and sets of alternative protocols for managing them. The taxonomy can form the basis of design handbooks for guiding the systematic solution of component integration problems. SYNTHESIS, a prototype software application development tool based on that perspective, has been developed and successfully used to minimize the manual effort required to integrate independently developed components into new applications.

1. Motivation

As the size and complexity of software systems grows, the identification and proper management of interconnection dependencies among various pieces of a system has become responsible for an increasingly important part of the development effort. In today's large systems, the variety of encountered interconnection dependencies (such as communication, data translation, resource sharing, and synchronization dependencies) is very large, while the complexity of protocols for managing them can be very high.

Dependencies among software components are especially important in component-based software development. In this case, the core functional elements of an application are implemented using off-the-shelf components. The focus of the design effort then lies in integrating these components by identifying and properly managing their interdependencies and mismatches. The practical difficulty of achieving wide-spread software reuse is a manifestation of the fact that component integration is not a trivial problem. Nevertheless, most current programming languages and tools have so far failed to recognize component interconnection as a distinct design problem which should be separated from the specification and implementation of the underlying components.

The distinct nature and equal importance of components and dependencies is captured relatively well in high-level, architectural descriptions of systems. In such descriptions components are typically depicted using boxes and dependencies using arrows. However, at that level of description dependencies are usually informal artifacts and their exact translation into implementation-level concepts is not obvious.

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 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 invocations, shared variables, etc. Such mechanisms are not sufficient for

Accepted to the Fifth International Symposium on Assessment of Software Tools and Technologies (SAST97), June 3-5, 1997, Pittsburgh, PA.

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 [13]. 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:

- *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 managing protocols in a number of different places inside a program. Therefore, in order to understand a protocol, programmers have to look at many places in the program. Likewise, in order to replace a protocol, modifications must be made in many different modules.
- *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 (often 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 [7]. In many cases this requires extensive code modifications or the introduction of additional code around the component. In most cases, such modifications are designed and implemented in an ad hoc manner.

Based on the previous observations this paper claims that, if we are to achieve large-scale component-based software development, we need new methodologies and tools which treat the interconnection of software components into new applications as a distinct design problem, entitled to its own representations and design frameworks. Such methodologies will be based on theories of component interconnection that organize and systematize the existing knowledge in the field of component integration, as well as facilitate the creation of new knowledge in the field.

To this end, Section 2 of this paper proposes a framework for studying software component interconnection. The framework is based on software system representations that provide distinct abstractions for components and their interdependencies. Such representations allow the systematic classification of different kinds of dependencies and associated coordination protocols into design handbooks of component integration, similar to the well-established handbooks that assist design in more mature engineering disciplines. Section 3 briefly reports on SYNTHESIS, a component-based software development environment based on our framework. Section 4 discusses related work. Finally, Section 5 sums up our conclusions and presents some directions for future research.

2. A Framework for Studying Software Component Interconnection

2.1 A Coordination Perspective for Representing Software Systems

One of the reasons behind the failure of today's programming languages and methodologies to recognize component interconnection as a distinct design problem is the lack of expressive means for representing interdependencies and their associated coordination protocols as distinct and separate entities from the interacting components. Therefore the first ingredient of our framework is a representation that achieves this distinction. The representation is based on the principles of coordination theory.

Coordination theory [12] is an emerging research area that focuses on the interdisciplinary study of coordination. One of the intended applications of coordination theory is the design and modeling of complex systems, ranging from computer systems to real-life organizations. Coordination theory views such systems as collections of interdependent processes performed by machine and/or human actors. Processes are sets of activities. Coordination theory defines coordination as the management of dependencies among activities. It makes a distinction between two orthogonal kinds of activities:

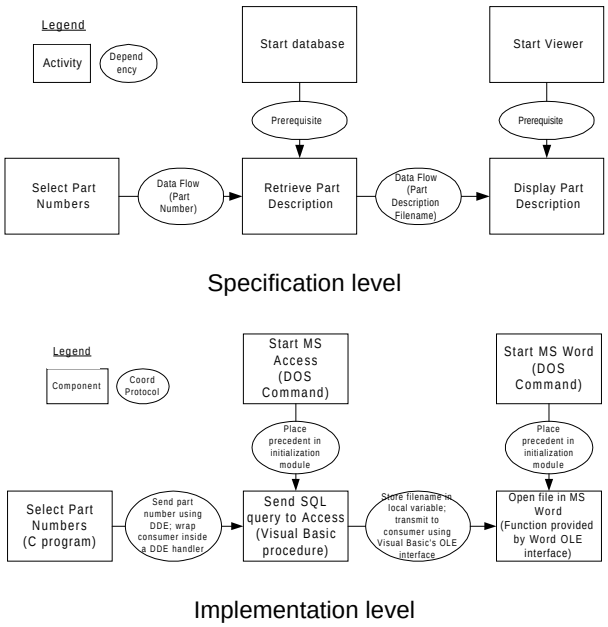


Figure 1: Representing a software application as a set of activities interconnected through dependencies.

- *Production (*or *core) activities*. Activities directly related to the stated goals of a system. For example, the SQL engine of a database system would qualify as a production activity in that system.
- *Coordination activities*. Activities which do not directly relate to the stated goals of a process, but are necessary in order to manage interdependencies among production activities. Algorithms that control concurrent access in multi-user databases would be considered coordination activities under this framework.

The above definitions suggest representations in which software systems are depicted as sets of interdependent software activities. At the specification level, activities represent the core functional elements of the system while dependencies represent their interconnection relationships and constraints. At the implementation level, activities are mapped to software components that provide the intended functionality, while dependencies are mapped to coordination protocols that manage them. Figure 1 depicts an example of a software system specification and implementation using such a representation.

2.2 A Design Handbook for Integrating Software Components

The existence of representations that treat dependencies and coordination processes as distinct entities enable the construction of taxonomies of software interconnection problems and solutions. This section presents the beginnings of such a taxonomy. The taxonomy contains the following elements:

- 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

Our taxonomy uses multi-dimensional design spaces to classify both dependencies and coordination protocols. It begins by identifying a small number of generic dependencies. For each generic dependency, it defines a number of design dimensions that can be used to further specialize the relationship. These dimensions form a design space that contains different specializations of the given dependency. Each point in the design space defines a different specialized dependency type.

Furthermore, for each dependency, our taxonomy identifies a few generic coordination processes that manage it. It also defines a design space that contains several related specialized versions of these coordination protocols. The dimensions of the design space are the questions the designer will have to answer in order to select one of the available coordination processes for managing a given dependency.

2.2.1 Overview of the Dependencies Space

An important decision in making a taxonomy of software interconnection problems 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 [11,12], 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, usability of the resource, 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.

It is not possible to complete describe the taxonomy in the limited space of this paper. Instead, the following sections will present a small subset of the taxonomy of flow dependencies, as well as an example of how it can be used to guide the design of software interconnection protocols. A full description of the taxonomy is contained in [3].

2.2.2 A Taxonomy of Flow Dependencies

Flow dependencies encode relationships among producers and consumers of resources. This section presents a generic model for classifying flow dependencies and a framework for designing coordination protocols for such dependencies. The framework is based on some results of coordination theory, extended and adapted for the field of software components.

Malone and Crowston [12] have observed that, whenever flows occur, one or more of the following sub-dependencies are present:

- *Usability*. Users of a resource must be able to effectively use the resource.
- *Accessibility*. In order for a resource to be used by an activity, it must be accessible to that activity.
- *Prerequisite*. A resource can only be used after it has been produced.

In the following paragraphs we will introduce dependency and coordination process design spaces for each of the lower-level dependencies. The design space for generalized flow dependencies is defined by the product of the design spaces of the component dependencies.

Usability Dependencies. Usability dependencies state the 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. 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 design dimensions listed in Table 1.

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

Table 1: Design dimensions of usability coordination protocols.

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 protocols depend on resource kind (see Table 3)*	

Table 2: Design dimensions of accessibility coordination protocols.

Producers-Consumers	*Generic Mechanism*	*Examples*
one-one	• Point-to-point channels	- OCCAM channels [8] - UNIX sockets
	• Pipes	- UNIX pipes
one-many	• Broadcast Calls	-ISIS Multicast [2]
many-one	• Asynchronous Calls	- Asynchronous message passing
	• Synchronous Calls	- Procedure calls - RPC - MS Windows DDE
many-many	• Broadcast Calls	- ISIS Multicast [2]

Table 3: Examples of transport protocols for data resources.

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.

There are two broad alternatives for making resources accessible to their users (Table 2):

- 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.

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.

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.

Table 4 shows four generic processes for managing prerequisite dependencies. Each generic process can be further specialized according to a number of design dimensions specific to the process. For example, peer synchronization can be specialized according to the type of event used for synchronization. Table 5 contains a partial list of events. For each event, different execution environments provide different sets of corresponding system calls, providing yet another level of protocol specialization.

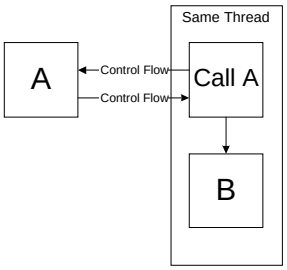
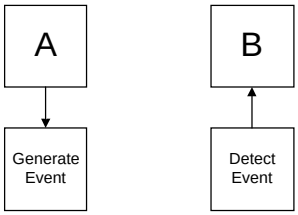
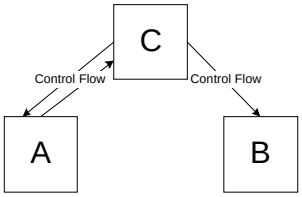
Producer Push		As soon as the precedent completes, it invokes the consequent by explicitly passing control to it.
Consumer Pull		Before it begins execution, the consequent synchronously calls the precedent.
Peer Synchronization		Both precedent and consequent are executed by independent threads of control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

Table 4: Generic processes for managing prerequisite dependencies.

Event type	*Generate*	*Detect*	*Reset*
Semaphore	Signal Semaphore (V)	Wait on Semaphore (P)	Reset Semaphore
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

Table 5: Examples of synchronizing events.

2.2.3 Designing Interconnection Protocols

This section will provide an example of how the framework can be used to guide the design of interconnection protocols among software components. Because only a small subset of the taxonomy is presented in this paper, the example will also by necessity be very simple.

Suppose we would like to connect two existing pieces of code: A C program providing a graphical interface that repeatedly asks the user for part numbers, and a Visual Basic program which queries a database and displays descriptions of the corresponding parts. The C program returns integer part numbers while the Visual Basic program expects strings. Figure 2 shows the components and their interconnection relationship, in this case a simple data flow.

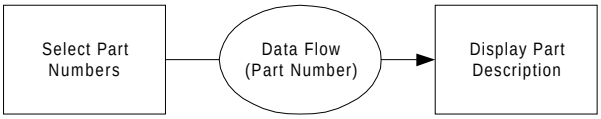
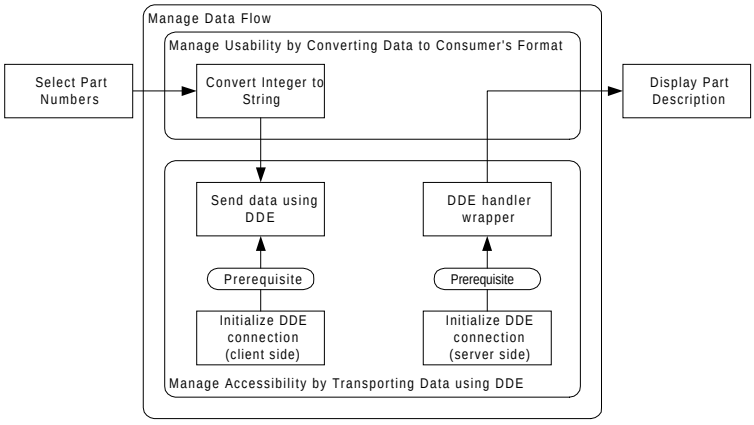


Figure 2: A simple software system.

According to our framework, in order to interconnect the two components, we need to design a coordination protocol for the data flow dependency. Following our generic model for flows, this means that we have to design protocols for managing usability, accessibility, and prerequisite dependencies.

To manage usability, we elect that the producer will be responsible for making the data usable to the consumer (see Table 1). In this example, this will require the addition of code at the C component for converting data from integers to strings.



C module:

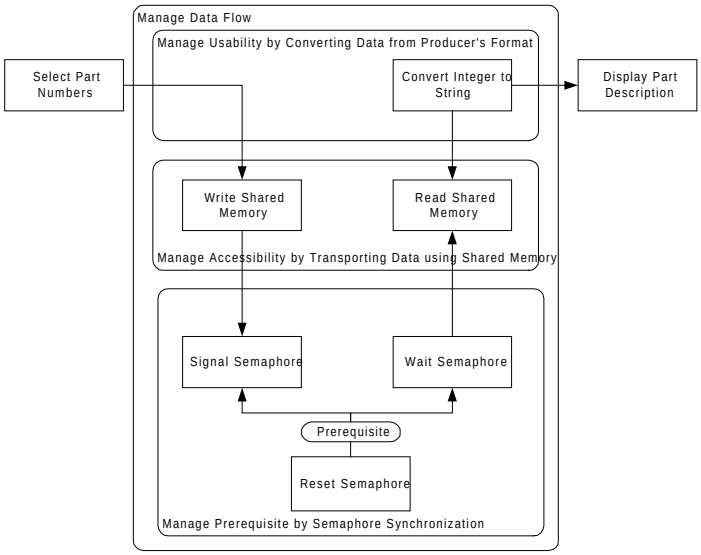
```
DDE_Init(&cli_id);
…
n = Select_Part();
s = itos(n);
DDE_Send(cli_id, s);
…
```

Visual Basic Module:

```
…
DDE_Init DDE_handler
…

Sub DDE_handler(s)
…
  Display_Part s
…
End Sub
```

Figure 3: One protocol for managing the data flow dependency of Figure 2.



C module:

```
Reset(sema);
…
n = Select_Part();
Write_Mem(loc, n);
Signal(sema);
…
```

Visual Basic Module:

```
…
Wait sema
Read_Mem loc, n
s = CStr(n)
Display_Part s
…
```

Figure 4: An alternative protocol for managing the data flow dependency of Figure 2.

To manage accessibility we first preclude the possibility of integrating the two components in the same executable, because they are written in different languages. We therefore have to transport the data from producer to consumer. Our framework provides a set of possibilities for doing this:

One possibility would be to use an RPC protocol to transmit the data from producer to consumer. DDE (Dynamic Date Exchange) is one such protocol supported by Microsoft Windows. The advantage of such a protocol is that it explicitly passes control from producer to consumer, thus managing the prerequisite dependency as well. The resulting protocol is depicted in Figure 3. In this protocol, the C component acts as a client, while the Visual Basic component is wrapped inside a handler for a DDE call and acts as a server.

Another possibility would be to use a shared memory location or a shared file, whose filename is fixed in advance and known to both parties. This solution would require us to address the prerequisite relationship separately: Make sure that the Visual Basic program only reads the next part number after it has been written by the C program. We select a peer synchronization mechanism specialized to use semaphores as the synchronization event. Finally, as shared memory locations are best for storing numbers, conversion from integers to strings is done at the consumer side. Our choices result in the protocol depicted in Figure 4. Notice that, in this protocol, the two components are eventually wrapped in two executables which run independently and synchronize implicitly[1].

In conclusion, our framework can not only guide the design of interconnection protocols in a systematic way, but also point out the range of alternatives available to the designer at each step.

3. The SYNTHESIS Application Development Environment

3.1 Overview

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 is devoted to a very brief description of the SYNTHESIS system. A detailed description can be found in [3].

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: 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).

A Design Handbook of Software Interconnection. A prototype version of a handbook of common software interdependencies and coordination protocols has been developed. The handbook is an on-line version of our taxonomy of dependencies and coordination processes. The design spaces of our framework have been implemented by hierarchies of increasingly specialized SYNOPSIS entities. For example, Figure 5 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.

A Design Process for Generating Executable Applications. SYNTHESIS supports a process for generating executable systems by successive specialization of their SYNOPSIS descriptions. The process automates the reasoning we used in Section 2.2.3 to design a coordination protocol for the flow

[1] The protocol for managing prerequisite dependencies shown in Figure 3 allows more than one part numbers to be generated before one of them is displayed. In this application such behavior would most likely not be acceptable. Reference [3] contains a taxonomy of different variations of prerequisite dependencies and corresponding coordination protocols that would give a fully satisfactory solution to this problem.

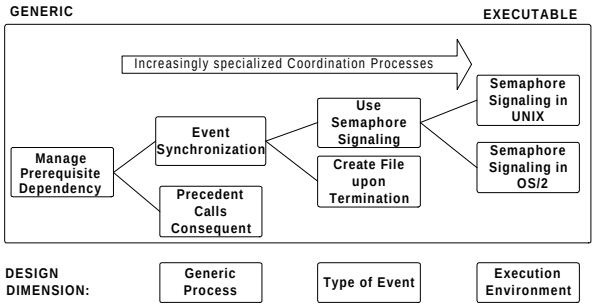


Figure 5: A hierarchy of increasingly specialized coordination protocols for managing prerequisite dependencies.

dependency and integrate our two components into a complete system. It 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 application element which is still not specific enough for code generation to take place (e.g. a dependency for which no coordination protocol has been specified):
   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 (e.g. it replaces the above dependency with a compatible coordination protocol for managing it) and recursively applies the same process to all elements in the decomposition of this element.
3. After all application elements have been replaced by implementable 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.

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.

3.2 Using Synthesis to Facilitate Component-Based Software Development

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
- 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 were very encouraging. 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. Related Work

4.1 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 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. 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.

4.2 Architecture Description Languages

Architecture Description Languages (ADLs) provide support for representing the high-level structure of software systems in terms of their components and their interconnections [9, 14]. They are an evolution of Module Interconnection Languages (MIL), first proposed in the '70s [5]. Most ADLs 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). 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. The project that comes closest to our work is UniCon [15].

4.3 CASE Tools and Software Design Assistants

A number of research tools attempt to facilitate the design and development of software systems by providing graphical, architectural views of systems and automated assistants which guide users through the design process. STILE [16] provides good support for graphical component-based design, but does not provide particular support for distribution or for managing component mismatches. The Software Architect's Assistant [10] is a visual environment for constructing distributed applications. Aesop [6] exploits the notion of architectural style to assist users in constraining their design alternatives and verifying the correctness of their designs.

Broadly speaking, SYNTHESIS also provides a graphical architecture description language and a design assistant for generating executable applications. However, the specific models (activities, dependencies, and coordination processes), relationships (decomposition, specialization) and design operations (replace dependencies with compatible coordination processes) supported by SYNTHESIS are different from the above systems and specifically geared to facilitate the integration of heterogeneous, multilanguage, and possibly incompatible software components. It will be interesting to see how good ideas from various software design assistants can be constructively combined.

4.4 Component Frameworks

Component frameworks such as OLE, CORBA, OpenDoc, etc. [1] and our coordination perspective were both motivated by the complexity of managing component interdependencies. However, the two approaches represent very different philosophies. Component frameworks enable the interoperation of independently developed components by limiting the kinds of allowed relationships and by providing a standardized infrastructure for managing them. Only components explicitly written for a framework can interoperate with one another.

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.

5. Conclusions and Future Directions

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 interconnection 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 relationships and coordination protocols for managing them can be usefully represented as independent entities, separate from the interdependent components. Furthermore, they can be systematically organized in a design handbook. Such a handbook can assist, or even automate the process of integrating a set of independently developed components into a new application.

Our experience with SYNTHESIS, a prototype application development environment based on these principles has demonstrated both the feasibility and the practical usefulness of this approach. Nevertheless, we view the work reported in this paper as only the beginning of an ongoing effort to develop better methodologies and tools for supporting component-based software development. Some areas we plan to address in the immediate future include:

- *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.

References

1. Richard M. Adler. Emerging Standards for Component Software. *IEEE Computer*, March 1995, pp. 68-77.
2. K.P. Birman, A. Schiper and P. Stephenson. Lightweight Causal and Atomic Group Multicast, *ACM Transactions on Computing Systems*, vol. 9, Aug. 1991, pp. 77-113.
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.
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. 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.
6. D. Garlan, R. Allen and J. Ockerbloom. Exploiting Style in Architectural Design Environments. *Proceedings, ACM SIGSOFT '94 Symposium on Foundations of Software Engineering*, December 1994.
7. 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.
8. Inmos Ltd. *Occam Programming Manual*. Prentice-Hall, Englewood Cliffs, NJ, 1984.
9. 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.
10. J. Kramer, J. Magee, K. Ng and M. Sloman. The System Architect's Assistant for Design and Construction of Distributed Systems. *Proceedings of 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, Sept. 1993, pp. 284-290.
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. 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.
14. 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.
15. 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.
16. M.P. Stovsky and B.W. Weide. Building Interprocess Communication Models Using STILE. *Proceedings, 21st Annual Hawaii Int. Conf. On System Sciences*, 1988, Vol.2, pp.639-647.