Understanding Software Automation

Ph.D. Thesis

April 2, 1996

Liam Hartmut Peyton

Institute for Electronic Systems

Aalborg University Center

Frederick Bajers Vej 7

Aalborg 9220

Denmark

The MirrorWorks Software Inc.

70-3205 Uplands Drive

Ottawa, Ontario

K1V 9T4

Canada

peyton@magi.com

Dansk resume

Denne afhandling er organiseret omkring en arkitektur, der giver et begrebsmæssigt fundament for forståelse af teknologien bag softwareautomatisering. Denne arkitektur er udviklet gennem en række projekter, inden for hvilke softwaresystemer blev udviklet til at automatisere vidensarbejde (forstået som informationsarbejde i modsætning til manuelt arbejde). Arkitekturen tjener det formål at give et videnskabeligt fundament, som kan bruges til at lede og informere praktikeren i brugen af og yderligere udvikling af teknologien til softwareautomatisering. Gennem præsentationen af arkitekturen vil vi også karakterisere beskaffenheden af det vidensarbejde, som automatiseres af software. Denne karakteristik er igen udviklet og forfinet gennem det projektarbejde, som denne afhandling er baseret på. Intentionen med afhandlingen er at lede og informere praktikere, efterhånden som organisationer tilpasser deres arbejdsprocesser til software automatiseringsteknologi.

Igennem vore projekter har vi observeret, at vidensarbejde er centreret omkring opgaver, som udføres med dokumenter. Vores hovedtese er, at vidensbaseret arbejde kan automatiseres ved at indlejre dets opgaver og dokumenter i software. I introduktionskapitlet vil vi forklare vores karakteristik af vidensarbejde og introducere tesen i forhold til vores arkitektur. Et gennemgående scenarium, som illustrerer vidensarbejde i en bank, vil blive brugt til at begrunde og illustrere vores karakteristik af vidensbaseret arbejde. Scenariet er en sammensætning og simplificering af faktiske situationer, som blev automatiserede i projekter for industrielle kunder på vegne af Knowledge Exchange Communications, og som er beskrevet i Appendiks A.2.

En undertese i dette arbejde er, at softwareudvikling er et eksempel på vidensbaseret arbejde, der kan automatiseres med vores arkitektur. Detaljerede eksempler fra to projekter, et system kaldet SoDA, som automatiserer softwaredesign (Appendiks A.6) og et system kaldet Axiant, som automatiserer database systemudvikling (Appendiks A.5), vil blive brugt som gennemgående eksempler på, hvordan arkitekturen kan anvendes. Disse to projekter er udvalgt, fordi SoDA var vores første forsøg på at implementere arkitekturen som et forskningssystem på Stanford Universitet, hvorimod Axiant brugte den samme arkitektur til at skabe et nyt værktøjssæt til udvikling af applikationer, som nu sælges verden over af Cognos Inc. Vores arbejde i andre industrielle projekter (Appendiks A.2, A.3, A.4) har bragt os i kontakt med de spirende standarder og arkitekturer til softwareautomatisering inden for områderne kontorsystemer, netværksstyring inden for telekommunikation og elektronisk udgivelse af dokumenter. Disse erfaringer er beskrevet i underafsnit om industriel praksis, og indgår i de sektioner om relationer til andet arbejde, der forefindes sidst i hvert kapitel.

Hovedtesen og arkitekturen er uddybet i kapitel to til fem. I kapitel to observeres det, at dokumenter bruges i vidensarbejde til at udveksle information, som bibringer viden til andre. En undertese er, at dokumenter kan kategoriseres i typer og defineres af modeller, som foreskriver deres indlejring i software. Derefter præsenterer vi en definition af de begreber, der bruges til at bygge dokumentmodeller, og vi diskuterer spørgsmål vedrørende deres implementation diskuteres. Projekteksemplerne i dette kapitel illustrerer mekanismer til at implementere kodegenerering, og understøttelse af semantiske relationer imellem og inden for dokumenter.

I kapitel tre viser vi, at et område inden for vidensarbejde kan identificeres ved typen af indgående dokumenter, og de opgaver der udføres med dem. En undertese siger, at en softwareproces, kaldet en dokumentagent, som giver adgang til dokumenter - defineret ved dokumentmodeller - gennem handlinger grupperet i opgaver definerer et område inden for vidensarbejde, inden for hvilket information kan udveksles. Derefter præsenterer vi en funktionel nedbrydning, af de egenskaber der kræves i en sådan proces, og diskuterer spørgsmål vedrørende design af en dokumentagent. Projekteksemplerne i dette kapitel illustrerer mekanismer til at styre samtidige opgaver, og til at styre delt adgang til dokumenter i en arkitektur med flere agenter.

I kapitel fire observerer vi, at udførslen af opgaver inden for vidensarbejde kan karakteriseres som problemløsning. Løsninger til tidligere opgaver kan genbruges, når nye opgaver skal udføres. En undertese

er, at arbejde kan automatiseres ved genbrug af løsninger, som er indlejrede i software som opgaveskabeloner. Derefter præsenterer vi en klassifikation af opgaveskabeloner baseret på den traditionelle metode til problemløsning inden for kunstig intelligens, og vi diskuterer nogle spørgsmål vedrørende integration af automatisering ved brug af skabeloner i vidensarbejde. Vore projekteksempler i dette kapitel illustrerer mekanismer til automatisering af constraint maintenance inden for design, og automatisering af fremstillingen af programmer i databasesystemer.

Kapitel fem er konklusionen på denne afhandling. Vi viser, at vidensarbejde finder sted inden for en bredere ramme af menneskelige institutioner. Løsninger skabt af deltagere bliver koordineret i en vidensarbejdsproces for at opnå mål inden for disse institutioner. En fuldstændig bearbejdning af dette ville bringe os uden for rammerne denne afhandling. Ikke desto mindre observerer vi, at der ser ud til at være en naturlig videreudvikling af vores arkitektur til at understøtte en fuldstændig vidensarbejdsproces. En undertese er, at vidensarbejdsprocesser kan indlejres i en speciel type dokumentagent, en opgavekoordinator, som koordinerer løsninger tildelt deltagere knyttet til andre dokumentagenter. Derefter præsenteres en funktionel opløsning af de egenskaber, der kræves af en opgavekoordinator, og vi diskuterer nogle implikationer af og begrænsninger ved vores arkitektur. Denne diskussion er udført i forbindelse med et tilbageblik på, hvad vi har lært gennem de projekter, der anvendes som eksempler.

Vore projekterfaringer har givet en række resultater, som vi tilskriver vores arkitektur. For det første blev der opnået store forbedringer i tidsforbruget på det vidensarbejde, der blev automatiseret af de systemer, som blev bygget. Derudover var der signifikante forbedringer i gennemførtheden og fuldstændigheden af det vidensarbejde, der blev udført og styret med disse systemer. Arkitekturen gjorde det også nemmere at styre opgaven med at bygge automatiseringssystemer. Der blev opnået store forbedringer i softwareudviklingstid, og der blev opnået store besparelser i vedligeholdsudgifter. Endelig var den udviklede software yderst konfigurerbar og flytbar.

Det er værd at bemærke, at der er visse antagelser, som skal være opfyldt, for at disse resultater kan opnås. Arbejdet skal kunne udføres via et computer medium. Den nødvendige hardware og software infrastruktur skal være tilgængelig. Arbejdet skal være standardiseret, sådan at præcise og fuldstændige dokumentmodeller kan defineres, og der skal være en rutine og gentagelse i arbejdet, som kan indfanges i opgaveskabeloner. Softwareautomatisering kræver en fundamental forståelse af det specifikke arbejde, der skal automatiseres, såvel som af den teknologi der anvendes.

Her er et resume af vores tese og dens underteser:

Vidensarbejde kan automatiseres gennem indlejring af dets dokumenter og opgaver i software

Softwareudvikling er et eksempel på vidensarbejde, der kan automatiseres ved vores arkitektur

Dokumenter kan kategoriseres i typer og defineres ved modeller, som er forskrifter for deres indlejring i software

En dokumentagent som giver adgang til dokumenter, defineret ved dokument modeller, gennem handlinger som er grupperet i opgaver, indlejrer et område af vidensarbejde, inden for hvilket information kan udveksles

Arbejde kan automatiseres ved genbrug af løsninger, som er indlejrede i software som opgaveskabeloner.

En vidensarbejdsproces kan indlejres i en opgavekoordinator, som koordinerer løsninger tildelt deltagere, der hører til dokumentagenter.

Preface

Almost everyone who read this document in its early stages asked me to clarify what academic field this work should be considered part of, who my intended audience was, and what exactly did I mean by the title. “Understanding” in the title should be interpreted as a gerund like seeing, in “seeing is believing”, while “software” should be interpreted as an adjective as in “the software industry”. This thesis describes my way of understanding automation that is achieved by the use of software. The question that remains, of course, is just what exactly are we automating with software? I think people should ask that question more often, especially in the context of justifying the effort to automate, and certainly before becoming embroiled in the details of how to automate. This thesis discusses the automation of “knowledge work”, by which I mean work which is concerned primarily with the communication and processing of information contained in documents.

It is probably overly simplistic to say that my intended audience is anyone who is interested in understanding software automation, but I will just add a few words about my approach in writing the thesis. I have attempted to provide a scientific foundation that can inform and guide the practitioner, while recognizing that not many in that audience will have the time or the inclination to read a Ph.D. thesis. My approach to understanding software automation is grounded in a conceptual characterization of knowledge work which is used to model its information processing aspects for the purposes of creating software systems. I know there is more to knowledge work than information processing, and my interest is in automation that improves the quality of work for people. However, it has not been practical for me to do much more than mention that from time to time in the thesis.

It should be clear from the last paragraph that I would like the ideas in this thesis to become part of the practice of building software automation systems for knowledge work. It should also be clear that my approach is strongly influenced by the Scandinavian approach to software development. I have worked with some of its more famous proponents (Kristen Nygaard, Bent Bruun Kristensen). This approach is well known in object-oriented programming for its emphasis on a conceptual foundation for programming constructs, and for its emphasis that software should embody models of the domain that is being automated. There is a central premise within the field which says that a domain can not be properly automated by software until the domain itself is well understood. This above all else rings true for me.

I hope that part of my contribution to the Scandinavian approach will be a new awareness of some techniques and ideas that I have brought with me from my background in artificial intelligence. While at Stanford University, I worked with some of its more famous proponents (Ed Feigenbaum, Cordell Green) as well as its critics (Terry Winograd). I have taken the theory developed in artificial intelligence to fully automate problem solving and tried to adapt it to help people work better by looking at automation in the context of an organization of people working with information on computers. I hope that I contribute to the field of artificial intelligence by creating more awareness of the manner in which automation can be integrated with human activity.

Acknowledgments

I have had the good fortune to follow a path through three different countries that has brought me in contact with many fine institutions and wonderful people. I have many good memories of the time I have spent in Aalborg, Stanford, and Ottawa. I would like to take the time to personally thank those particular institutions and people who have affected the course of my work on this thesis.

Aalborg University not only provided a wonderful working environment during my studies there, but through a combination of programs was able to provide me with a teaching assistantship, living expenses, and travel expenses from North America. The courtesy and professionalism of the Department of Computer Science and Mathematics in accommodating my inability to speak Danish was very much appreciated. I would also like to thank the government of Denmark for the programs they have established to encourage foreign research. Not only did they make my stay in Denmark possible, but they funded the Danish researchers I worked with at Stanford University before coming to Aalborg. Denmark has created a research community of international renown, that I am honoured to have been a part of during my thesis work.

The National Science and Engineering Research Council of Canada provided me with a postgraduate scholarship for five years, and gave me special permission to use it outside of Canada. Stanford University provided me with research or teaching assistantships that supplemented my scholarship. I am especially grateful for an emergency grant that the Department of Computer Science provided for me, during my first semester at Stanford when the timing of my support did not coincide well with the timing for tuition payments. The Center for the Study of Language and Information was the source of money for my research assistantships during my first years at Stanford, while the Knowledge Systems Laboratory was the source in my final years at Stanford. The Kestrel Institute provided a research assistantship one summer, and for many years was very generous in providing time, equipment and instruction to support my work.

A number of companies have been most supportive in providing an environment that both provided essential practical experience and encouraged my desire to experiment and innovate. Cognos paid my moving expenses when I returned to Ottawa. They allowed a woolly headed academic to roam far beyond his official duties, took the time to learn from my ideas, and taught me the difference between theory and practice. Dataware Canada had the vision to combine my research interests with the interests of their clients. When I left to form my own company, they provided encouragement and good advice. The MirrorWorks Software has been able to support me, thanks to the flexibility and support of my clients Bell Sygma and Knowledge Exchange Communication. They have been very accommodating in arranging my contracts in order that I might find the time to write this thesis.

On any path to understanding, one encounters teachers along the journey that show the way. I have been blessed by encounters with great teachers whose ideas have revolutionized my thinking. My first semester at Stanford, Terry Winograd gave a course on the Phenomenological Foundations of Computers and Cognition that revealed to me the nature of communication and automation. Kristen Nygaard revealed to me the nature of software systems and the perspectives particular to them when I was a teaching assistant for his course on Object-Oriented Programming (including everything my mother didn’t tell me). My last semester at Stanford, I was a teaching assistant for Ross Quinlan, on his seminar in applying Expert Systems Technology that revealed to me the nature of change for organizations which tried to introduce software automation. Nick Stodin revealed to me the nature of knowledge work and the process by which businesses reach objectives in a series of tutorials and seminars over the past few years - and challenged me to automate it. Bent Bruun Kristensen revealed to me the nature of my thesis when he taught me how to write it. I have been continually amazed at what has tumbled out of my work, now that I have learned to use the tools he provided me with.

I have had the pleasure of working with many individuals on several projects over the years. The projects and the people are listed in the appendix. I would like to make special mention of a few people. Terry Winograd, Cordell Green and Penny Nii were very supportive of me at a time in my work when I did not know where I was going. Raymonde Guindon built the diagrammer interfaces for the SoDA system and pushed me to improve the interaction that SoDA could provide. Ed Feigenbaum convinced me of the importance of applying my work to real situations in industry. Brad Cameron and Celine Goyette conceived the idea of a code generator for the document models in Axiant, and performed the lion's share of the implementation for Axiant's document agent. Their patience as I grasped for vocabulary to explain my ideas was only outdone by their professionalism and wizardry in finding ways to make them real. Colin Moden proposed the original idea of a language sensitive editor and program creation assistant for Axiant. Our interaction throughout pushed me to continually innovate and improve the quality of automation that was achieved. Jamie Winterhalt made real for me both the people and the technology that was part of automation in the Clientship Toolworks system on which the scenarios in this thesis are based. I also profited from many discussions with John Kambanis.

Friends and family have been especially supportive during my long journey. I could not have returned to Ottawa to work at Cognos without Stewart Winter who interceded on my behalf on more than one occasion during the hiring process while I was in California. Kasper Osterbye not only went to great lengths to make my work at Aalborg possible, he provided for every aspect of our stay in a country where we did not speak the language. There are few memories of time spent in Denmark that do not include him. And through every twist and turn, my family has been there. No one knows better than my parents Noel and Ursula, my sister Ulrike, my wife Susan, and Susan's mother Helen how stubbornly erratic I can be. They have managed to gracefully accept my actions, questioning me only enough to be sure that I understood what I was doing - even if no one else did. Susan has been beside me every step of the way and always manages to put events into perspective, separating the essential from the absurd.

In the end, understanding is achieved through dialogue. This thesis came to fruition in a most remarkable series of sessions in 1995 in which I was debriefed, transformed, and hung out to dry by Kasper Osterbye and Bent Bruun Kristensen. Kurt Normark provided a fresh eye that confirmed the results when I put pen to paper after those sessions. While we do not share the same mother tongue, there is an understanding between us that transcends words. Our dialogue over the years has nourished this thesis with a constant supply of food for thought. Tak for mad[1].

[1] A Danish expression which means literally "thanks for food". It is customary in Denmark at the end of a meal to thank those who have prepared it.

Table of Contents

List of Figures

Chapter 1 Overview

This thesis is the culmination of ten years research and experimentation towards developing better software tools and environments. During that time the nature of computing has changed dramatically. The changes have been enabled by increased performance and storage capacity, as well as the introduction of pc-based workstations, graphical user interfaces, local and wide area networks. The software we create today has become increasingly more complex. As a result, the tools we use to create software must become more sophisticated in order to manage that complexity.

The initial insight that drove this research was that software development could be automated by building explicit models of software in terms of its components and by applying artificial intelligence techniques that incorporated those models. While building several systems based on this idea, a new understanding of software began to form. Developers build software by building documents that describe how software will behave. We also noticed that in many organizations the tasks performed with documents were now being performed electronically. This work that centered around documents was often referred to as *knowledge work*. We realized that knowledge work, including software development, could be automated by building explicit models of electronic documents and by applying artificial intelligence techniques that incorporated those models.

An understanding of knowledge work has also formed in our research concerning the role of software automation and the changes and opportunities it creates. Knowledge work involves the communication of information in a form that is useful to other people. More than a data processing tool, the computer has become a medium for the communication of information [Winograd & Flores 85] in which much of our daily work (and play) takes place. Electronic documents bring information to life. They embody information in a way which can respond to the actions of people. When knowledge work tasks are performed in an electronic medium, then the results created by those tasks can become information that is used to guide and automate.

Two basic principles have driven this research. The first principle is that software is a model of an information process in some application domain. To automate information in a domain we must have an in depth understanding of that domain [Madsen et al 93]. Software tools need to support the process of modeling, because better models will form a basis for more complex software. Models define the electronic documents that we interact with to communicate information. When documents models are implemented in a software framework, we create a computer medium in which people can work.

The second principle is that automation is enabled by capturing and reusing the solutions that are created in a specific application domain. The power of automation is founded on the knowledge and experience of the domain reflected in those solutions [Buchannon 82]. Knowledge work needs templates to capture solutions that can be reused, because useful templates will form a basis for more productive and better quality work. When task templates are implemented in a software framework, we automate what is already known in our work.

The result of this research is an architecture for software automation. By this we mean, an architecture for understanding and implementing the automation of knowledge work done by people using electronic documents. Automation in this context, should be understood as the creation of software that enhances and extends the capability of people to perform work that has become increasingly complex. The reflections in this thesis on the nature of software technology and the nature of knowledge work provide a framework in which to continue understanding software automation.

1.1 Thesis Outline

This thesis has been organized around an architecture that provides a conceptual framework in which the technology of software automation can be understood. The architecture has evolved over the course of a number of projects in which software systems were built to automate knowledge work. It is intended to provide a scientific foundation which can be used to guide and inform the practitioner in using and further developing the technology of software automation. In presenting our architecture, we will also characterize the nature of knowledge work that is automated by software. Again, this characterization has been developed and refined through the project work on which this thesis is based. It is intended to guide and inform practitioners as organizations adapt their work processes to software automation technology.

We have observed in our projects that knowledge work is centered around tasks performed with documents. Our main thesis is that knowledge work can be automated by the embodiment of its documents and tasks in software. In this overview chapter, we will explain our characterization of knowledge work and introduce the thesis in terms of our architecture. A knowledge work scenario situated in a bank is used to motivate and illustrate our characterization of knowledge work in the overview chapter and throughout the rest of the thesis. That scenario is a composite and simplification of actual situations which were automated in projects for industrial clients on behalf of Knowledge Exchange Communications, Inc. that are described in Appendix A.2.

A subthesis of this work is that software development is one example of knowledge work that can be automated by our architecture. Detailed examples from two projects, a system called SoDA that automated software design (Appendix A.6) and a system called Axiant that automated database system development (Appendix A.5), will be used throughout this thesis to show how the architecture can be applied. These two projects have been chosen because SoDA was our first attempt to implement the architecture as a research system at Stanford University, while Axiant used the same architecture in creating a new application development toolkit which is now sold worldwide by Cognos Inc. Our work in other industrial projects (Appendix A.2,A.3,A.4) has brought us in contact with emerging standards and architectures for software automation in the areas of office systems, telecommunications network management, and document publishing. These are described in a section on Industrial Practice that is part of a section on Relationship to Other Work that appears at the end of each chapter.

The main thesis and our architecture is elaborated in detail in chapters two through five. In chapter two, we observe that documents are used in knowledge work to exchange information that conveys knowledge to others. A subthesis is stated that documents can be categorized into types and defined by models which are a prescription for their embodiment in software. We then present a definition of the concepts which are used to build document models and discuss some of the issues in implementing them. Our project examples in this chapter illustrate mechanisms for implementing code generation and for supporting semantic relationships between and within documents.

In chapter three, we observe that a domain of knowledge work can be identified by the types of documents used and tasks that are performed with them. A subthesis is stated that a software process, called a document agent, which provides access to documents defined by document models through actions that are grouped into tasks, embodies a domain of knowledge work in which information can be exchanged. We then present a functional decomposition of the features required in such a process and discuss some of the issues in designing a document agent. Our project examples in this chapter illustrate mechanisms for managing simultaneous tasks, and for controlling shared access to documents in an architecture with multiple agents.

In chapter four, we observe that, in knowledge work, the performance of tasks to achieve some result can be characterized as problem solving. Solutions to previous problems can be reused when performing a new task. A subthesis is stated that work can be automated by reusing solutions that are embodied in software as task templates. We then present a classification of task templates based on the traditional model of problem solving used in artificial intelligence and discuss some issues in integrating automation into knowledge work using templates. Our project examples in this chapter illustrate mechanisms for automating constraint maintenance in design, and for automating the creation of programs in database systems.

Chapter five is the conclusion of this thesis. We observe that knowledge work takes place within the larger context of our human institutions. The solutions created by participants are coordinated in a knowledge work process to achieve the objectives of those institutions. A full treatment of this would take us beyond the scope of our thesis. Nonetheless, we observe that it appears that a natural evolution of our architecture could be extended to support an entire knowledge work process. A subthesis is stated that a knowledge work process can be embodied in a special type of document agent, a task coordinator, which provides coordination of solutions assigned to participants associated with other document agents. We then present a functional decomposition of the features required in a task coordinator and discuss some of the implications and limitations of our architecture. This discussion is done in conjunction with a retrospective of what has been learned in our example projects.

Our project experience has shown a number of results which we attribute to our architecture. First, order of magnitude improvements were achieved in the time taken for knowledge work that was automated by systems that we built. In addition, there were significant improvements in the consistency and completeness of the knowledge work that was performed and managed with those systems. The architecture also made the task of building automation systems more manageable. Order of magnitude improvements in software development time and maintenance costs were realized. And the software which was created was highly configurable and portable.

It should be noted that there are assumptions which must hold before these results can be obtained. It must be possible for the work to take place in a computer medium. Appropriate hardware and software infrastructure must be available. Work must be standardized so that precise and complete document models can be defined, and there must be routine and repetition in the work that can be captured in task templates. Software automation requires a fundamental understanding of the particular work that is being automated as well as the technology that is being used.

Here is a summary of our thesis and subtheses:

> Knowledge work can be automated by the embodiment of its documents and tasks in software
>
> > Software development is one example of knowledge work that can be automated by our architecture
> >
> > Documents can be categorized into types and defined by models which are a prescription for their embodiment in software.
> >
> > A document agent, which provides access to documents defined by document models through actions that are grouped into tasks, embodies a domain of knowledge work in which information can be exchanged.
> >
> > Work can be automated by reusing solutions that are embodied in software as task templates.
> >
> > A knowledge work process can be embodied in a task coordinator, which provides coordination of solutions assigned to participants associated with document agents

1.2 Knowledge Work

"...

communicate	to convey knowledge of or information about
...	
communication	a process by which information is exchanged between individuals through a common system of symbols, signs, and behavior

..."

Webster's Ninth New Collegiate Dictionary (Springfield Ma., USA: Merriam Webster Inc. 1987), p. 266

1.2.1 The Vision

Computer systems create a communications medium in which information can be created and exchanged [Winograd 85]. More specifically, computer systems provide an environment which enables *the exchange of information between individuals through a common system of symbols, signs, and behaviors for the purpose of conveying knowledge*. Computer systems are based on data processing and data communications hardware. It is our ability as humans to attach meaning to symbols in the form of data that creates information. Knowledge is conveyed when the organization of that information is in a form which enables people to apply it in their work (or play).

Knowledge work is based on communication in the form of documents. Human institutions use documents, and create standards for their form and content, to structure information into a usable form. Knowledge work is broken down into tasks which involve the creation, modification and distribution of information in documents. Understanding knowledge work in terms of the relevant documents and the tasks which use them is the first step in automation.

Figure 1 is an illustration of what takes place when software is introduced into organizations to automate knowledge work. Participants in a domain of knowledge work organize their work into documents and tasks. The documents used are modeled (implicitly or explicitly) in order to create software that captures the data stored in documents and the actions performed in tasks. Software must provide an interface that people can interact with, as if they were performing tasks with documents. It must also provide an implementation in terms of data and actions that resides in computer hardware. Actions that take place outside of the electronic medium must be captured by recording them in documents.

This is the first level of automation that software can provide. It is our thesis that further levels of automation can be accomplished by explicitly embodying document models and tasks into a document agent. In Figure 1, the architecture of the document agent reflects the manner in which knowledge work is organized and performed. The actions performed by the document agent as solutions are created can be captured and defined in terms meaningful to the participant. We will use task templates to characterize solutions created in the performance of tasks. This will enable a second level of automation in which tasks are automated and managed by reusing the solutions defined by task templates.

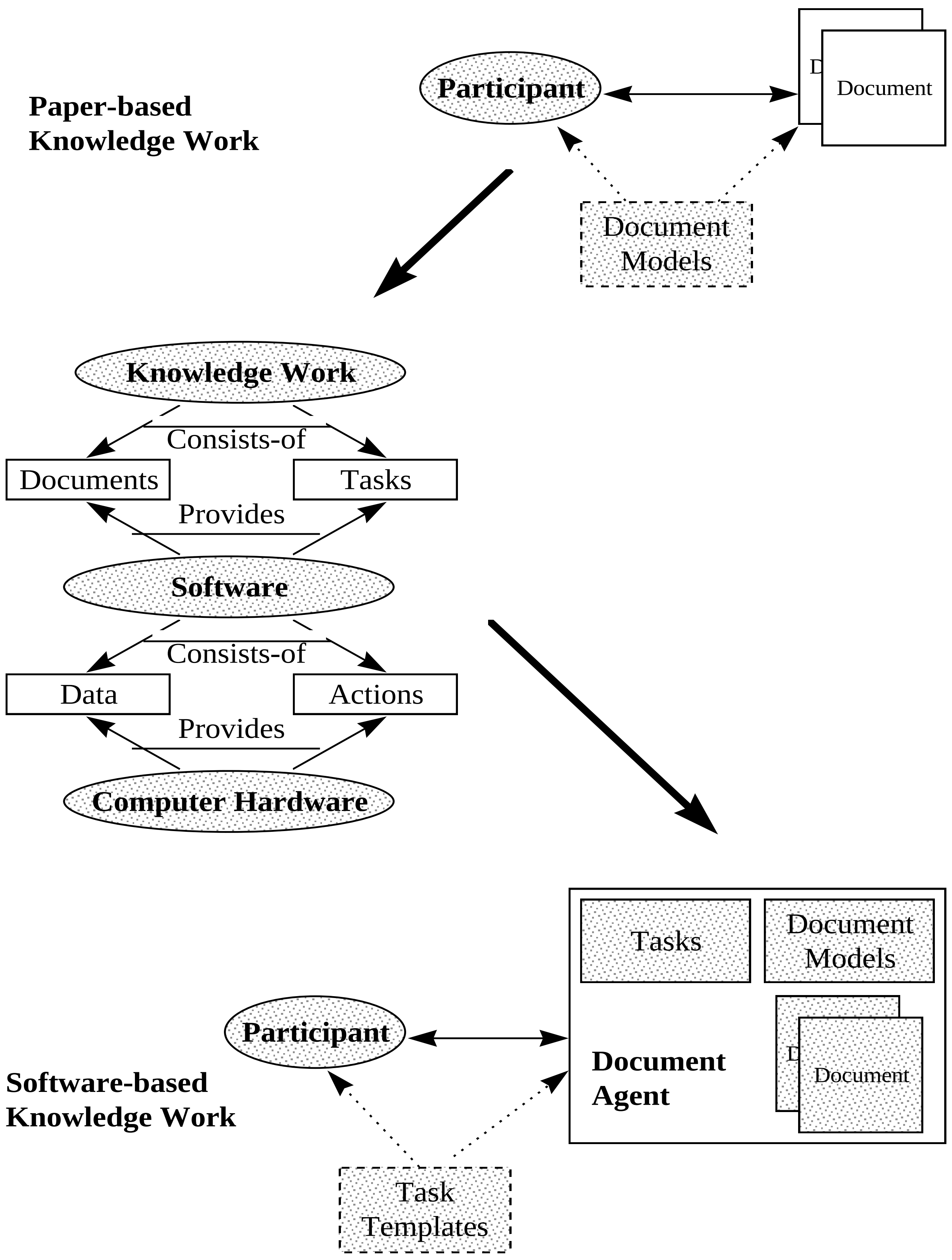


Figure 1. Software Automation of Knowledge Work

1.2.2 Scenario

Let us consider a scenario which is a composite and simplification of actual situations that were automated in projects for industrial clients on behalf of Knowledge Exchange Communications Inc. as described in Appendix A.2. Our intention in presenting this scenario is to capture the current state of the art in software automation and a little beyond in order to motivate our understanding of knowledge work and the architecture which we will present. There are many serious issues and difficulties which are simplified and ignored in the scenario in order to make the presentation straight forward. These issues will be examined later during the presentation of our architecture and in detailed examples from other project work. We will revisit this scenario in each chapter of the thesis.

In this scenario, we have a banker with a potential client. The banker would like to propose a portfolio of accounts, investments and debt instruments that will appeal to the client as a good way to arrange his money. In order to do this, the banker will need to collect information both to identify the client, but also to understand what the client's current money situation is. The banker will also need a catalog of the products that the bank provides which might be applicable to the client's situation (savings accounts, credit cards, mortgages). The banker will select from this catalog to create a proposed portfolio of products and present it to the client. If the client agrees, the banker will collect information and have the client authorize a request to implement the portfolio on behalf of the client by creating accounts, transferring money etc.

The documents and tasks involved in this process are diagrammed in Figure 2. Client information is collected in a standard data collection form we have labeled Client Form. Standard data collection forms, Order Forms, are also used to record any orders which result from the client accepting the proposal. There is a Product Catalog, which lists all the products the bank sells, organizing by type, and including relevant information about each product (cost, terms etc.). We have also shown a Client Portfolio document separate from a Proposal Letter. The Client Portfolio lists the bank's products that are proposed for the client. The Proposal Letter might be used to show this information in a more presentable and formal matter, which may also include information about the bank and the client to present a more complete picture.

To introduce software automation, we start with the documents. It will be natural to create a system which has graphical data entry forms for the Client Form, and for the Order Forms. The graphical forms have actions associated with them. Not only do they allow the banker to enter the information collected, but they can validate and assist the banker as the information is entered. For example, the software can make sure that the phone number entered is a valid number (no letters, a local prefix). The operations for editing the form can be more powerful as well. If the client has two similar assets (say two cars), once the information for one car has been entered, it can be duplicated (copied and pasted). Finally, once the information has been entered, actions associated with the form can be used to automatically file it (usually in a database). Since filing is automatic, retrieval is automatic. Once the client is entered in the system, any information in the Client Form that is also required in an Order Form, could be retrieved and filled in automatically.

The Product Catalog would also be filed in the database as a report that the banker could browse. The Client Portfolio is another data entry form. To fill it in, the banker needs to find products in the catalog which are applicable based on the information in the Client Form. Usually, there are business guidelines indicating the sort of client each product is intended for. If the guidelines are represented appropriately, then the software can select the products which apply. Once the portfolio has been created, the Proposal Letter is written using a word processor. If the form of the letter has been standardized, except for specific details taken from the Client Form and the Client Portfolio, then it can be generated automatically. The banker would only need to use the word processor for customizations. Finally, the software could assist the implementation of Order Forms. In simple cases, like opening a savings account, the system will perform the necessary actions automatically. In a complex case, like a mortgage, the form can be routed to the necessary departments (to do credit checks, appraisals, closing, etc.).

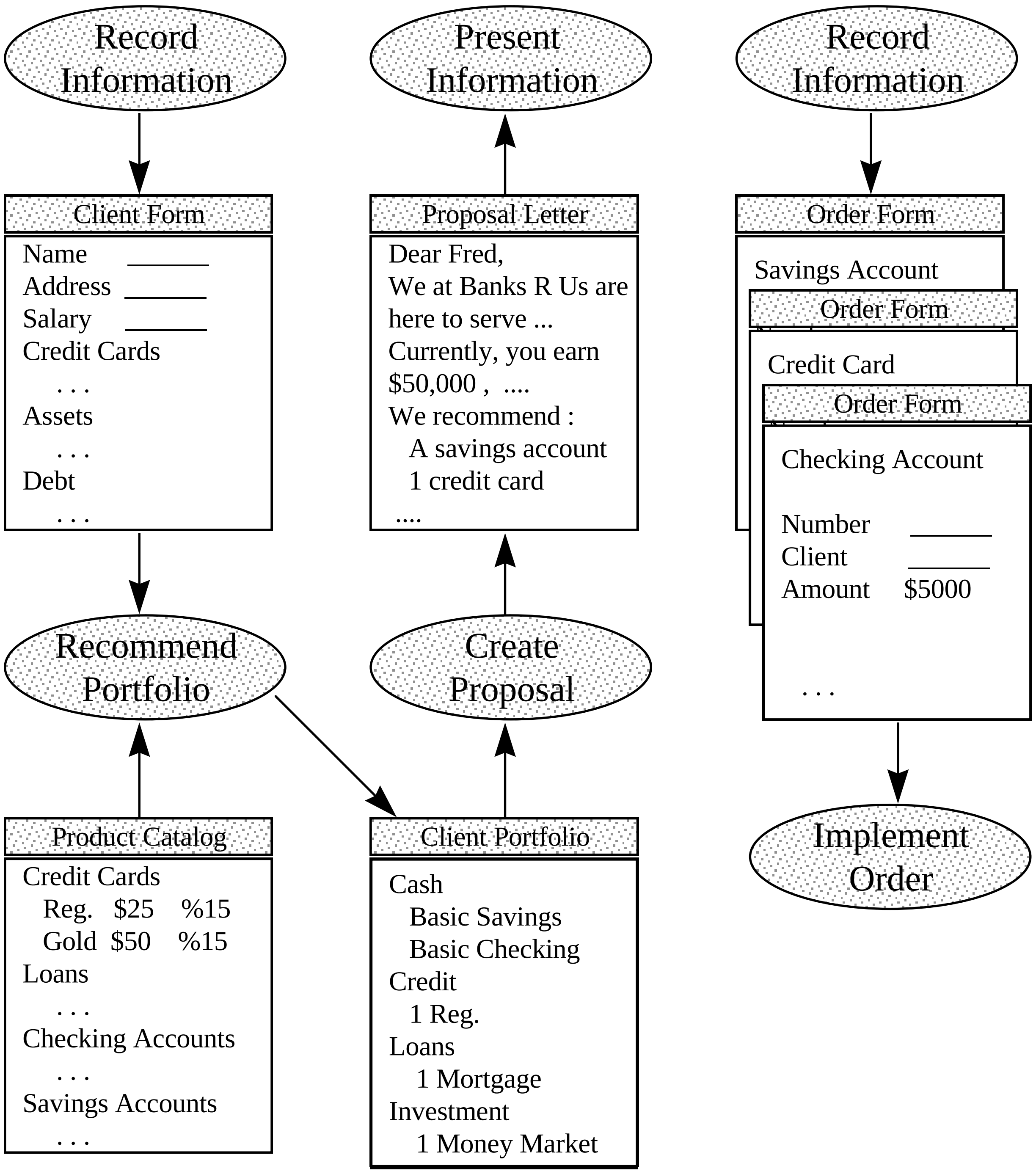


Figure 2. A Knowledge Work Scenario

Some observations can be made about the software that automates knowledge work in this fashion. The electronic documents used by the banker organize data into information and enable that information to be exchanged. It provides basic actions for editing documents as well as storing and retrieving them. Those basic actions can be enhanced with mechanisms for validation, cut and paste, linking one document to another, and the transfer of documents within the system.

The actions provided to the banker can be grouped according to the tasks the banker performs in his knowledge work. There are a particular set of actions that the banker performs to Record Information, Recommend a Portfolio, or Create an Order. Those tasks can be defined to reuse knowledge captured from previous solutions. When the banker needs to write a Proposal Letter, an existing standardized letter can be copied and then modified to customize it to the particular situation. The guidelines the banker uses to Recommend a Portfolio, capture the knowledge that determines when a product should be recommended.

The software also provides an interface to electronic documents that needs a framework to ensure the integrity of the system and the work being done. The banker should be able to undo edit operations, or cancel entire actions. Several bankers should be able work at the same time while the system ensures that their actions to not interact. Should something happen to the system (a power failure, or mechanical flaw), the system should be able to recover all the information that has been created in documents.

It is worthwhile to observe the human interaction and the emergence of new possibilities. The system reflects and records human judgments but does not replace them. No purchase is possible without the authorization of the client. The banker is responsible for the relationship with the client, including the collection and presentation of information and the authentication of client authorization. The application of business guidelines may be automated, but the guidelines and the software reflect bank policy for which the bank is responsible. At the same time, automation can be used to collect and analyze information that can improve the service provided by the bank (and presumably the success of the bank). If the proposal is rejected, the reasons for rejection can be recorded. That information can be automatically collected and collated with other rejections by those responsible for creating bank products and business guidelines.

1.3 An Architecture for Software Automation of Knowledge Work

The architecture for software automation of knowledge work that we present here starts with an understanding of the role played by documents in knowledge work. The exchange of information, i.e. communication, is fundamental to knowledge work. Documents record and standardize the information that is used and created in knowledge work. A domain of knowledge work can be delimited by identifying the types of documents used by participants in the domain, and the types of tasks which are performed by participants with each type of document. The actions used in performing tasks cause information to be exchanged: from participant to document (by creating or modifying information in documents), from document to document (by using and reorganizing information from other documents), and from document to participant (by reading and reacting to information displayed by documents). It is precisely this exchange of information that can be automated by software.

Software that automates knowledge work by automating the exchange of information creates a domain of information exchange. To the extent that work can be characterized by the exchange of information in documents, it can be automated by software. There will always be aspects of knowledge work which will exist outside of the electronic domain created by software. There will also always be aspects of software which will exist outside of the domain of knowledge work. The overlapping ellipses at the top of Figure 3, illustrate the point. Software automation matches a domain of information exchange to a domain of knowledge work. Where the two ellipses overlap, automation can occur. Participants must adapt in order to integrate work that remains outside the domain of information exchange, and to accommodate aspects of software that are outside of their domain of knowledge work.

1.3.1 Architecture

In our architecture, software that will create a domain of information exchange is built as a document agent (shown in Figure 3). Data within the document agent is organized into structures that can be presented to participants as documents. A document agent will usually interact with a graphical display or some other client process through which participants can access the document agent (and the documents which it controls).

A document agent performs actions that exchange information on behalf of those participants. The actions are grouped within the document agent according to the tasks that a participant performs in the domain of knowledge work. Task actions exchange information by accessing data in documents through their interaction with document models. Each type of document is defined by a document model. A document model determines the organization of data within a document as well as the primitive actions a task can use to access data.

The tasks which a participant performs will often involve a series of actions performed by a document agent on a number of documents. Task templates define previously created solutions that can be reused by the document agent eliminating the need for a participant to request the same series of actions over and over again. A document agent will usually interact with a facility for persistent storage or some other server process to store and retrieve the data contained in documents. There may be other functionality, as well, that a document agent provides to participants through its interaction with other server processes.

There are, of course, other aspects of the document agent not shown here which will be needed to manage its activities. Requests and responses from clients and servers need to be coordinated. The document agent must also coordinate its actions on behalf of participants to ensure the integrity of the work that is performed. Among other things, active documents and their association with the data in persistent storage must be tracked.

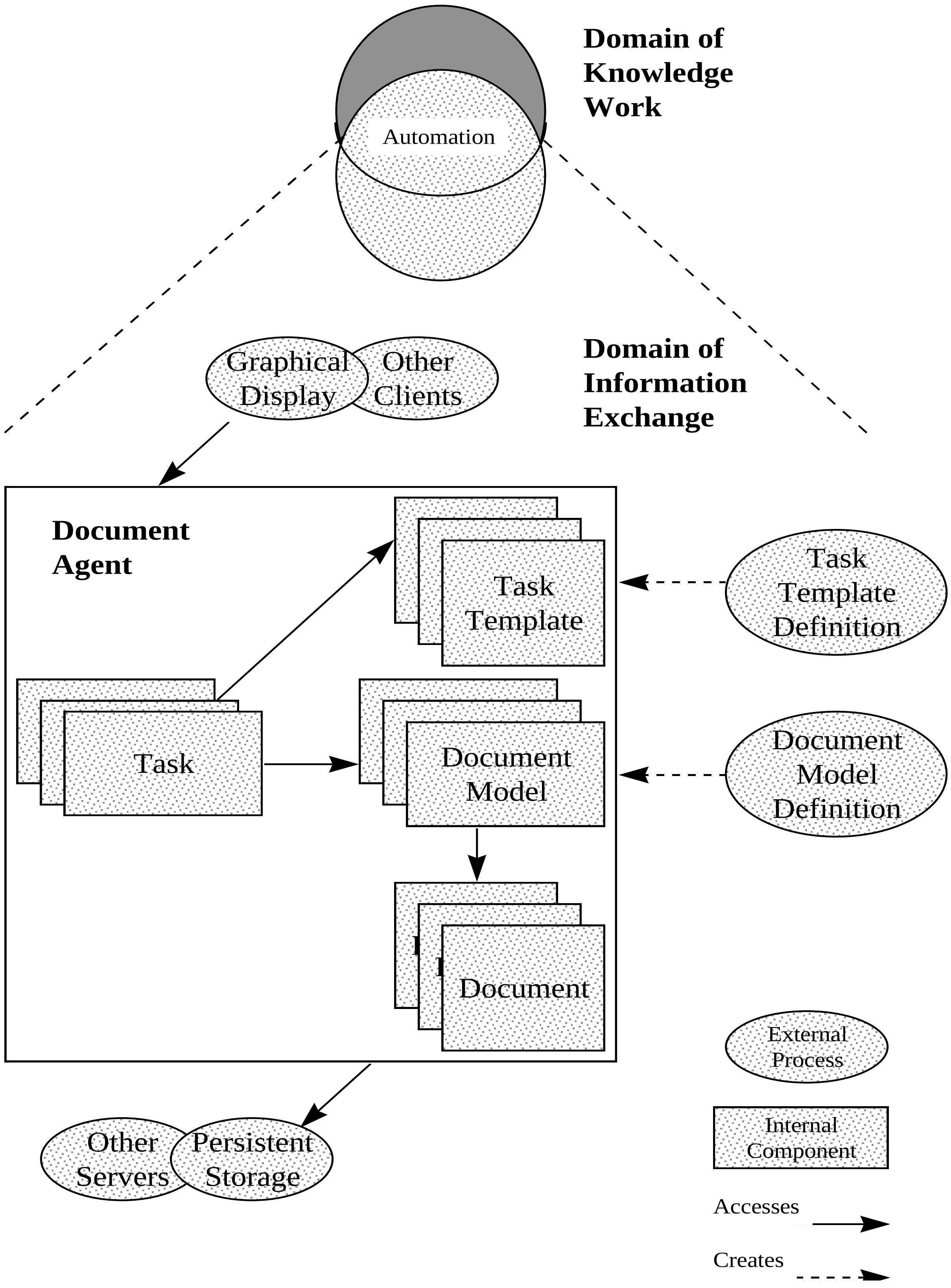


Figure 3. Document Agent Architecture

There are a number of advantages to this architecture. It reflects the domain of knowledge work. Data is grouped into documents. Documents are modeled according to type. Actions are grouped into tasks. Templates are defined to reuse previous work. Because it reflects the domain of knowledge work, it is easier to understand how the domain of information exchange created will map onto the domain of knowledge work. That makes development of the software that implements the document agent easier to manage. As change occurs in the domain of knowledge work, it will also be easier to maintain the software. In addition, the architecture provides flexibility in configuring the operation of the document agent within a domain of knowledge work. The document agent can work with a variety of client, and server processes without affecting the arrangement of tasks and documents within the document agent. The operation of the document agent can also be configured without changing the basic architecture by replacing or adding tasks, document models, and task templates.

The most fundamental aspect of this architecture is that it is based on explicit conceptual models. The data in documents is interpreted by participants as information that conveys knowledge, because it reflects our conceptual understanding of some part of the world. A document groups data and captures relationships between those groups in a manner that is analogous to the manner in which humans form concepts and learn relationships between concepts. Our document models will define the organization of a document and the actions that can be used to exchange information in terms of groups of data, and relationships between those groups. The advantage of this approach is that it reflects the conceptual understanding of participants within the domain who use documents to communicate information. Documents designed this way should be more effective in a domain of knowledge work. Another advantage is that once a document model has been defined, well established software techniques can be used to generate the software needed to implement a document model within a document agent.

Document models are also the basis for the automation that can be achieved through the use of task templates. Task templates describe reusable solutions in terms of the initial documents used, the final documents that resulted and the actions that were used. An advantage of this approach is that it reflects the conceptual understanding of participants who create solutions. They are the ones who will be the most effective in determining which solutions are most appropriate for reuse. The definition of a task template is then a matter of generalizing or standardizing a solution in a manner that maximizes the effectiveness of its reuse in the performance of a task. The actions of a task can be implemented so that they organize and reuse templates as they are defined. In this manner, the focus of participants is directed toward the acquisition and application of what is known in the domain of knowledge work.

1.3.2 Knowledge Work Automation Spectrum

We characterize the automation that is achieved in our architecture by the level of interaction it supports for participants as they perform work. Electronic technology by itself would only support an interaction in terms of electronic signals. Computer hardware provides infrastructure that supports the communication and processing of data. Software implemented as a document agent is intended to raise the level of interaction experienced by participants from that of interacting with data to that of interacting with information. Ideally, participants would be performing their work at a level of interaction which allows them to focus on their knowledge and understanding in the domain of knowledge work. When a participant is interacting at the level of data it requires adjustment and interpretation by the participant to perceive the information that can be extracted, and then another level of interpretation in order to perceive the relationship of that information to their knowledge of the domain.

Figure 4 situates our architecture in a knowledge work automation spectrum. People are capable of operating at any level of interaction. Communication between people involved in knowledge work exchanges information that conveys knowledge. In other words, communication happens at the level of information, but in a manner that is intended to enable participants to interpret it at the level of their knowledge and understanding.

Machines enhance the amount, speed and distance over which communication can take place, but often at the expense of reducing the level of communication. A computer permits a data level of interaction by converting signals into symbols like letters, numbers, or colors on a graphical display. People see the symbols as data about some aspect of the world without consciously perceiving or having to interpret the signals which create them.

Document models are a description of how people organize and structure data so that it can be immediately perceived as information that reflects their conceptualization of the world. The intention is to eliminate the need to consciously perceive and interpret the data which creates information. A document agent which displays documents to participants and performs actions based on a document model is providing an information level of interaction. Ideally, the participant is no longer conscious of the operating system or the data which makes the behavior of the document agent possible. Note, though, that this is depends on the degree to which a document model is able to capture the organization and structure of data that communicates information to participants.

Task templates are a description of solutions that can be immediately perceived as knowledge to be applied to the performance of tasks. A document agent which reuses templates in its actions can provide an interaction that reduces the amount of interpretation required by a participant to communicate at the level of knowledge and understanding. In chapter five, we will consider the possibility of a special type of document agent, a task coordinator, which can be used to define and coordinate the solutions that are created by participants performing knowledge work. A task coordinator enables a knowledge level of interaction by enabling a participant to define the problems to be solved and monitor the results that are reported. That interaction is interpreted by the participant in terms of their understanding of the overall objectives that a human institution has for the domain of knowledge work.

We have found this spectrum a useful device for categorizing the software automation of knowledge work, especially for focusing on the nature of interaction between people and machines that is expected to result. We have not considered its relevance to other forms of automation. Nor have we considered its relevance for automation of areas other than knowledge work.

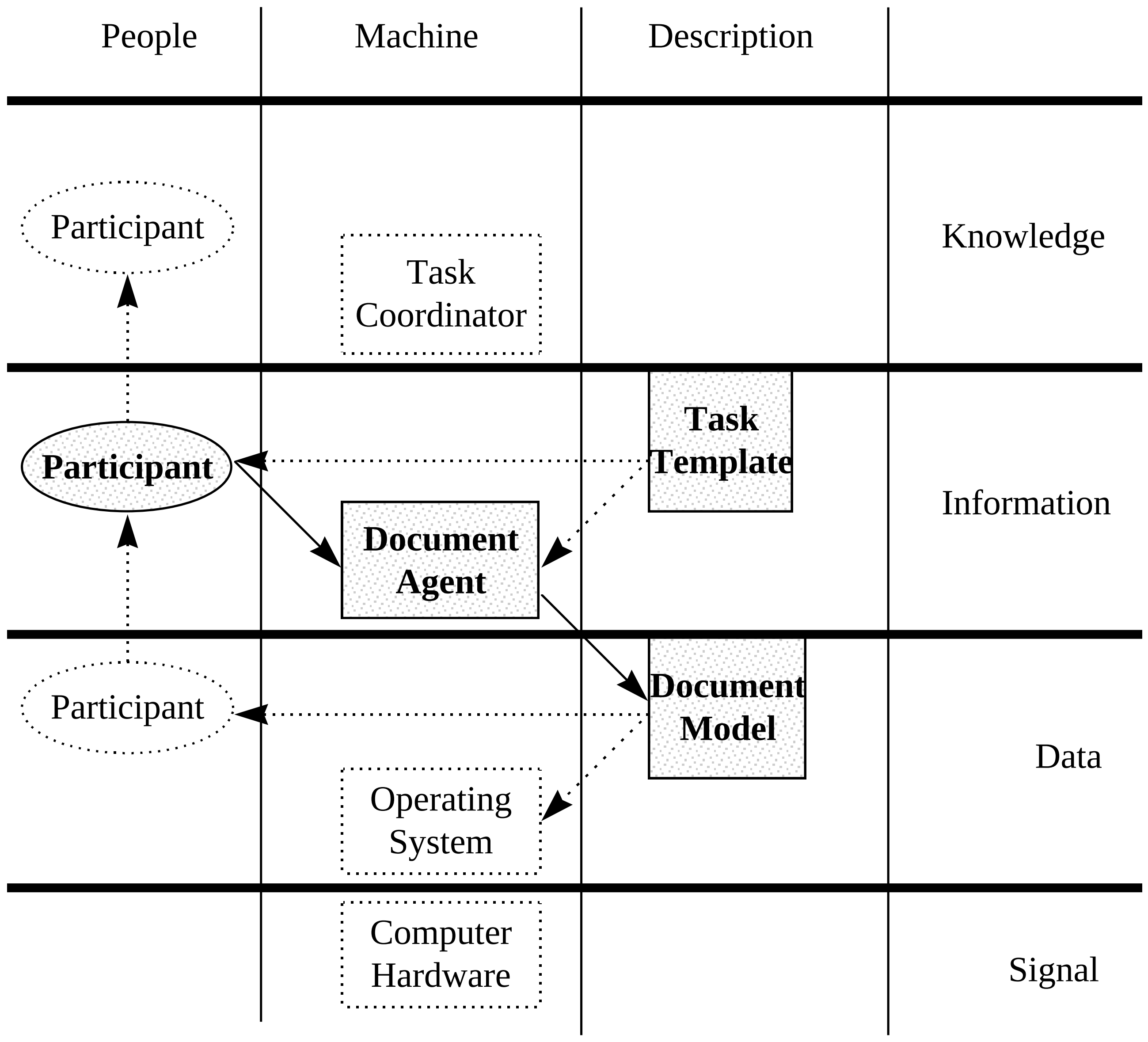


Figure 4. Knowledge Work Automation Spectrum

1.4 Projects

The architecture presented in this thesis has been developed based on experiences in a number of automation projects. These projects and the participants are described and acknowledged in the appendix. For the purposes of exemplifying the architecture and its practical application, we will concentrate on examples from two projects in the areas of software design and software development. A brief description of the projects and the architecture of the systems that were implemented is included here to give some background for the examples that will be used throughout the thesis.

One of the contributions of this thesis is the characterization of software design and development as knowledge work. Software design and development centers around documents that describe and define the system which is being built. However, software developers do not build software systems, at least not the way a carpenter might build a house. They build documents, which for all intents and purposes are the software system. A computer processes those document so that it behaves exactly as the document said it would. Software designers also build documents. The difference between a design document and a development document often seems to be one of degree. Development documents can be generated from design documents when a design document contains enough detail.

Software design and development are domains of knowledge work which are particularly appropriate for software automation. To a large degree, the work is already taking place in a computer medium, so there are fewer issues in terms of adapting the domain of work to accommodate automation. Automation can focus on raising the level of interaction provided to participants.

1.4.1 A Software Design Assistant (SoDA)

The work we describe here was performed during our participation in the KASE (Knowledge-assisted Software Engineering) project (Appendix A.6). KASE was a joint project between the Kestrel Institute and the Knowledge Systems Lab at Stanford University. The aim of the project was to create a software engineering environment where the types of expert systems software developed at the Knowledge Systems Lab could be designed and assembled from a library of software components. The approach was to adapt the technology of program transformations [Smith, Kotik, & Westfold 85] developed at the Kestrel Institute for implementing formal specifications and incorporate graphical user interfaces which showed different design views of the system being assembled. Blackboard systems[Nii & Aiello 79] were the first type of expert system addressed by the KASE project.

I implemented a system called SoDA (Software Design Assistant) which provided automated support for the design of software systems. There were three objectives in the work on building SoDA. The first objective was to create a conceptual model of software systems in terms of their component parts that would support design diagram views, and which would define constraints on the relationships between components that must hold true for a design. The second objective was to develop design transformations analogous to program transformations which could be used to modify the design while maintaining the validity of the design in terms of its constraints. The third objective was to support the style of interaction that was most suitable for software designers in their work. This was based on a study of software designers [Guindon 90] that had been performed by a cognitive psychologist associated with the project, Raymonde Guindon. Raymonde also implemented the graphical user interface for one of the design diagram views supported by SoDA.

SoDA supported design as knowledge work that centered around the creation of design diagrams. A design diagram is a graphical representation that shows a view of certain relationships between software components for some part of a system. Usually, there is an intended or actual correspondence between a design diagram of a system and the source code which implements it. In case tools, there is often a code generator which partially or fully generates source code out the back end and a reverse engineering component which can construct diagrams from the source code. Some common views of software are control flow (which shows the flow of activation between components) and data flow (which shows the flow of information between components). One of the main requirement for design diagrams is that they define a system that is both complete and consistent. It is also important to be able to manage the complexity of the system being designed. Mechanisms are needed for browsing flexibly through different parts of the system and switching easily between different types of views.

Figure 5 shows the SoDA system in terms of the document agent architecture that we diagrammed in Figure 3. The main type of document supported was a System Design document. A Design Session document was supported to track actions organized by activity that were performed during design. SoDA also included facilities for displaying and editing Source Code documents which implemented the design. The System Design documents was modeled as a hierarchical structure of modules and data with relationships between these components to capture dependencies associated with control flow and data flow. The models were defined and implemented as object classes in the KEE language which was used to implement SoDA.

SoDA’s document agent supported three clients: a Control Flow diagrammer, a Data Flow diagrammer and a Design Assistant. The first two clients were graphical user interfaces which presented different views of the software design document. The Design Assistant was a graphical user interface which managed the design session on behalf of a designer, tracking the design actions which were performed and their effects on design constraints. SoDA’s document agent tracked requests for actions made by the three user interfaces and provided a history mechanism which provided a very flexible undo feature. The document agent also read and wrote design documents to the file system (which served as the repository). Document links to source code associated with the design were provided. Systems were implemented in Lisp, so source code was displayed (and edited) in a ZEmacs Editor.

The main task supported by SoDA was provided in actions that changed the design of a system. SoDA also supported an automated task that tracked the effects of design changes by checking constraints associated with control flow and data flow. That task used task templates for constraint checking defined in KEE classes as constraint monitors that captured the knowledge of what constraints should be checked when changes occurred. Any violations were placed on a To Do list of problems to be resolved. The Design Assistant was also notified as violations were noticed. Another task supported by SoDA was providing assistance to the designer when fixing violations. The task used task templates for fixing violations which were defined in KEE classes as suggestion rules that captured the knowledge of what actions could be taken to fix a violation. The Design Assistant would display the list of suggestions at the request of the designer.

Document models for System Design and Design Session coupled with task templates for checking constraints and fixing violations proved very effective in automating design. Order of magnitude improvements in quality and productivity were demonstrated compared to diagramming tools without models or templates. In one instance, a blackboard system design was imported into SoDA that was considered complete and consistent by experts associated with the KASE project. Thirty constraint violations were flagged, resulting in substantial additions and changes to the design. Guided by SoDA, these were made in less than two hours.

On the other hand, the attempt to create design transformations analogous to program transformations was largely unsuccessful. The most useful transformations seemed to be the basic edit operations of copying and pasting components from other parts of the blackboard system and then making minor changes. It was felt that a large collection of sample components and sample systems coupled with templates specific to blackboard systems (rather than software in general) would provide the most effective support of design.

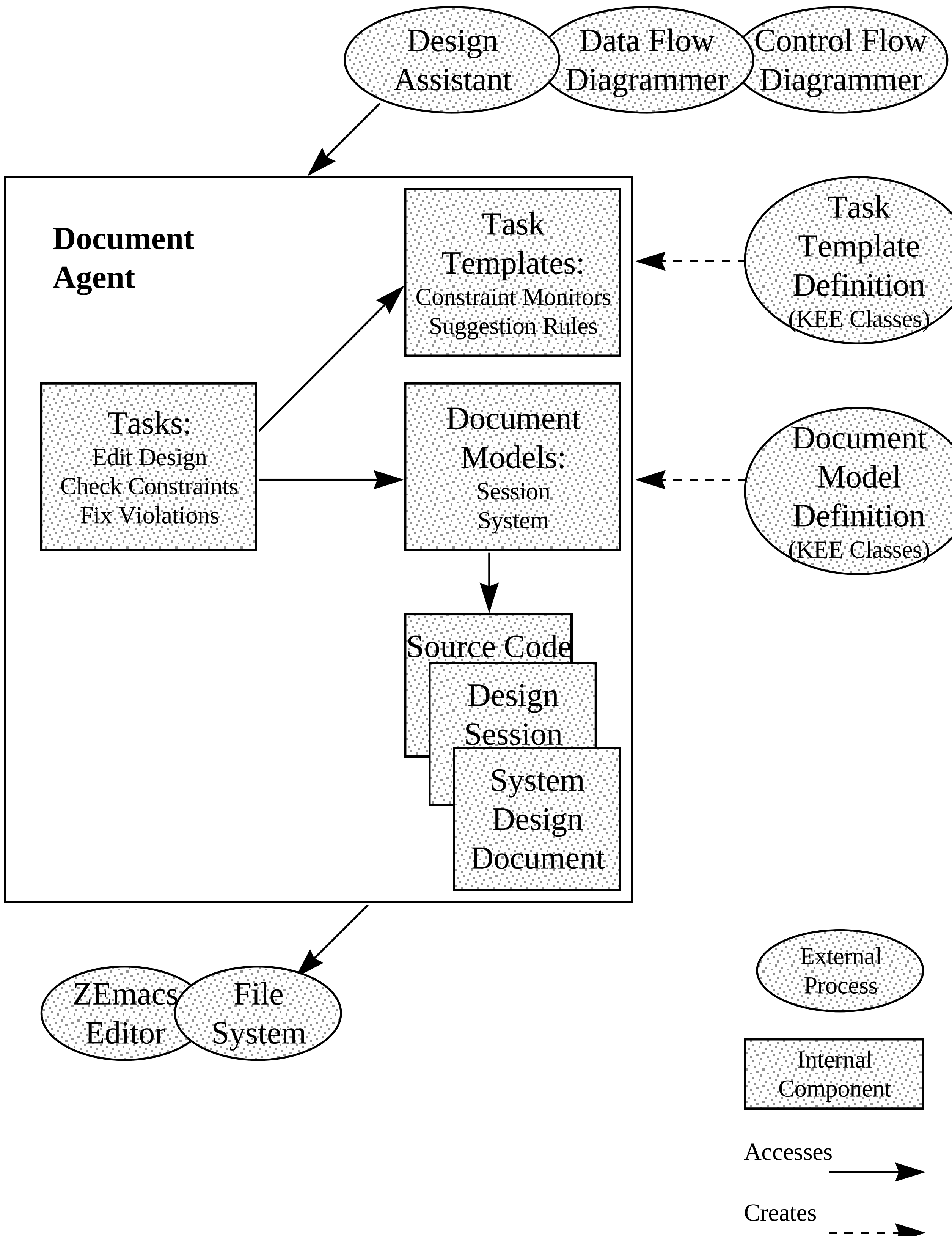


Figure 5. SoDA Architecture

1.4.2 A Software Development Environment for Database Systems (AXIANT)

The work we describe here was performed during our participation in the Axiant project (Appendix A.5) at Cognos Inc. The aim of the project was to provide a graphical, object-based environment for building database systems using the PowerHouse language. Flexible support for coordinating teams of developers working together was required, and it was considered vital that the environment automate and support rapid development of database systems deployed in a server, standalone pc, or client-server configuration. PowerHouse was one of the world's first fourth generation languages when it was made commercially available by Cognos in 1979. Axiant 1.0 for Windows 3.1 was released into production world wide in 1994.

As a Senior Software Engineer, I was one of the original team of seven developers and architects which took Axiant from its initial concept through to its full implementation and first beta release. I designed and developed the initial implementation of a Creation Assistant and a Module Designer which automated and supported rapid development through the use of task templates. I also designed and participated in the implementation of a document agent based on document models that supported object-based graphical interfaces and shared access to documents. It must be emphasized, though, that this was a team effort. I have mentioned people in the Appendix and in the acknowledgments whose participation was fundamental to work that I did and the ideas that are presented in this thesis. That list is by no means complete.

We will refer throughout to the "Axiant Prototype", and not Axiant 1.0 to make it clear that we are not describing the commercial product[Cognos 94]. We will present only selected aspects of the Axiant Prototype which are of direct interest to this thesis. We have also taken liberties in changing and hiding a number of details in order to simplify our presentation and protect trade secrets of Cognos Inc.

The Axiant Prototype supported data base system development as knowledge work that centers around documents. There are several documents which are produced in building a database system. There are database scheme documents which defines the types of data that can be stored: domains (for common pieces of information like date, money, address) and tables (for the different types of records like employee, customer). There are also interface documents which define the mechanisms for users to access data: forms (for viewing and updating records in a database), reports (for listing and summarizing the contents of the database) and utilities (for doing maintenance and whole scale changes to the database).

Figure 6 shows the Axiant Prototype in terms of the document agent architecture that we diagrammed in Figure 3. There were document models which modeled each type of database system document (Tables, Domains, Forms, Reports, and Utilities) as a collection of objects. A diagrammer and code generator was built to define the models and then generate the code (written in C) which implemented each model inside the document agent. A Session document was supported to track actions performed during development. A Repository document organized and supported the sharing of documents in a central repository.

The document agent interacted with several clients and servers. The Application Browser listed documents in the central repository. The Creation Assistant automatically generated Forms and Reports based on templates and input from a developer. The Database Designer created and maintained tables and domains in the targeted database. The Form Designer and Report Designer created forms and reports. The Module Designer edited PowerHouse code associated with forms, reports and utilities. The document agent tracked requests for actions made by clients, supported undo and enabled recovery in the case of a system failure. It also read and wrote documents to the file system (which served as the repository). Since documents in the repository were shared between developers, version control was provided by support for third party configuration management systems. In addition to the file system, communication links were maintained with a remote server in order to create databases and compile code that would run on a VAX or UNIX server.

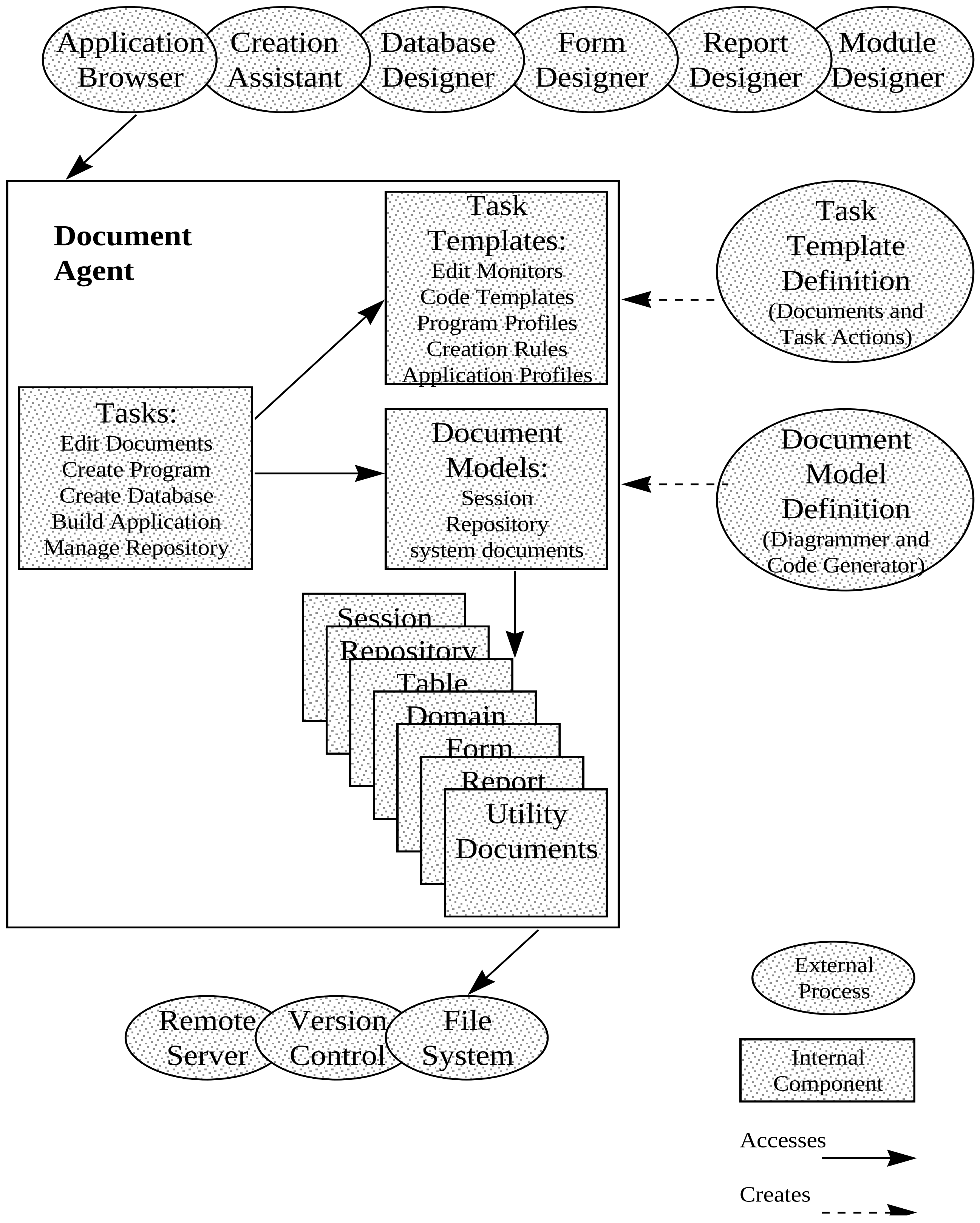


Figure 6. Axiant Prototype Architecture

The main tasks supported by the system were editing documents, creating programs, creating databases, building applications and managing the repository. Editing documents was supported by actions that changed the objects in documents. Special support was provided for the modification of objects which contained PowerHouse code. Task templates in the form of Code Templates were used, which could be selected and pasted in the Module Designer. Edit Monitors which captured the knowledge of when and how the code should be redisplayed enabled the Module Designer to color code and format the code in an appropriate style as changes were made.

Creating programs was supported by an automated task that automatically generated a Form or Report. That task used task templates in the form of Program Profiles that characterized common Forms and Reports, and Creation Rules that captured the knowledge of how Form and Report documents should be generated from a Program Profile. Creating databases and building applications was supported by actions that transferred files to a remote server, communicated instructions to create and maintain databases and compile programs, and return the results of those operations. Task templates were used in the form of Application Profiles that characterized common configurations and defaults for creating databases and building applications. These included such things as the type of database being targeted (relational, flat file), and the type of application being built (standalone pc, VAX or UNIX server, or pc client with VAX or UNIX server)

Managing the repository was supported by actions to group documents in the repository into directories according to the databases and applications in which they were used. There were also actions to control access to documents by different developers so that ownership could be determined and conflicts avoided. Special control was provided to ensure that all the changes made to documents in a session were committed to the repository as a single update.

Document models proved very effective in supporting graphical, object-based interfaces, and the Module Designer and Creation Assistant were effective in automating rapid development. A running application could be created in minutes instead of hours or days. Program and Application profiles also ensured that database systems were created which were consistent and complete in both their implementation and their look and feel. The document agent architecture was able to provide flexible support that enabled developers to work together. However, it was realized during the project that coordination of developers was best provided by hooks to support third party products for configuration management and project management. It also became clear that while objects were essential for graphical interfaces, persistent storage of documents would have to be in ASCII format for compatibility reasons with those third party products.

The architecture used in developing Axiant provided configurable and portable software. Document models could be added, replaced or changed without any impact in the processing of the document agent. At the same time, document models were protected from changes to client and server processes. An initial version of Axiant used a proprietary binary format for storing documents to file, a later version stored documents in a relational database, while the final version stored documents as ASCII files. In each case, the task Manage Repository was rewritten in a manner of days without any changes to other tasks or to the document models. The use of a code generator and diagrammer realized order of magnitude savings in development and maintenance costs. In an initial prototype, two document models took two developers two months to build in an inconsistent and error-prone fashion. With the code generator, models were built and tested in a matter of days by a single developer.

1.5 Relationship To Other Work

This thesis has evolved through experimental work done in several projects some of which were oriented towards research and others which were oriented towards practical applications in industry. As a reflection of that, we will discuss the relevance of our thesis work with respect to an understanding of software automation as it is articulated in research work, and as it is applied in industrial practice.

This thesis incorporates ideas from a broad cross section of work in both software engineering and artificial intelligence, but we will not attempt to provide an in depth survey of these fields. We will simply identify representative work which influenced us, or which could be considered as an alternative to our approach. Our remarks will serve to outline the context in which we understand each representative work, and will not attempt to provide a complete characterization of the work and all the issues which it addressed.

1.5.1 Research

Our thesis has been organized around three key ideas which we will relate to other work. The thesis develops an understanding of software automation based on a conceptual characterization of knowledge work that models its information processing aspects for the purposes of creating software systems. That understanding is presented in the form of a software system architecture that defines a model-based environment for supporting knowledge work. We also demonstrate how another level of software automation can be achieved within such an architecture by identifying and reusing templates for solving commonly occurring problems.

Software Automation

Our approach to software automation characterizes the domain that is being automated in terms of document models and tasks. There are several approaches to software automation which are based on a characterization of the domain that is being automated.

Object-oriented programming takes the approach of characterizing the domain that will be automated in terms of objects. Simula [Nygaard & Dahl 81] in the 1960's and Smalltalk [Goldberg and Robson 94] in the 1970's introduced the concepts and programming constructs which are commonly used today to characterize and define the domain which will be automated as well as to build the software that automates it. One normally distinguishes [Madsen et al 93] between object-oriented analysis in which an understanding of the domain is captured, object-oriented design in which a description of the software system is defined, and object-oriented programming in which the software is implemented in a programming language. A number of books discuss issues of methodology in this approach to software automation [Booch 94] [Jacobsen et al 92][Rumbaugh et al 91] [Coad & Yourdon 90] [Shlaer & Mellor 88].

Object-orientation is more general than our approach which is focused strictly on knowledge work. In addition, we identify a particular architecture which is most appropriate for automation of knowledge work. Our approach is compatible with object orientation in that document models and tasks can be analyzed, designed and implemented using object oriented techniques. Our understanding of the issues in matching a domain of information exchange to a domain of knowledge work have been strongly influenced by the Delta project [Haanlykken & Nygaard 81]. Development methods for building object-oriented systems described in [Jackson 83] and [Mathiesen et al 95] are similar to our method of separating document models, which define a domain, from tasks, which group functionality that can be performed in the domain. Our understanding of document models has been strongly influenced by work on conceptual analysis of language constructs for object oriented languages [Kristensen & Osterbye 94].

Traditionally, automation of knowledge work has usually centered around database systems with an approach that characterizes the domain in terms of data models. Entity-relationship modeling techniques [Chen 77] are used to define data models. There are a variety of methodologies and case tools which support this approach to software automation. [Martin 89] provides an introduction to the area. Fourth generation languages [Martin 85] have been used to automate the construction of database systems by associating data models with language constructs specific to data processing activities. Document models in our architecture can be defined and implemented using data modeling techniques. This can be important when the physical storage facility for documents is a database.

There are some recent initiatives [Goldfarb & Rubinsky 90] which define formalisms for modeling documents, as well as some initiatives to extend the capabilities of databases to support textual data stored in documents. The emphasis has been on defining a standardized storage format for documents that will be supported by software that processes documents, and to standardize document models so that software can support model-based interactions.

Artificial intelligence has been concerned with creating software that provides complete automation of tasks normally performed by people. Expert systems [Buchannon 82] usually combine a knowledge representation of the domain [Woods 75] [Winograd 75] in which tasks are performed with a representation in the form of rules of the knowledge used to perform a task. [Waterman 86] is a good introduction to the methodology of this approach to software automation. [Slagle & Wick 88] characterizes the type of tasks and domains where such automation can be effective. [Peyton 93] summarizes a set of guidelines applicable to knowledge work. Our approach uses the basic definition of problem solving [Amarel 68] found in artificial intelligence as a starting point for our automation of tasks using task templates. We adapt this to our context of support for people working interactively with documents. [Nanard et al 93] characterizes this as a generalist system as opposed to an expert system.

In our approach to software automation, we have been focused on supporting documents which are used to communicate information. In this we have been influenced by [Winograd & Flores 86] which discusses some limitations of the artificial intelligence approach and promotes the computer as a communication medium which can coordinate and support the communication between people which is essential to knowledge work. [Winograd 86] also articulates a language-action perspective on knowledge work which can be used as a basis for automation. [Stodin 91] discusses how a knowledge work process can be organized in terms of its communication in order to achieve strategic objectives for a business. Their work has not addressed the automation of individual tasks or the creation of document models which has been the focus of our work. They do define a context in which our work can be understood. We consider this context in our discussion of a task coordinator in chapter 5.

Model-based Environments

Our architecture uses a document agent to create a configurable model-based environment to support and automate knowledge work. We have been influenced by work that has taken a model-based approach to creating environments that support software development.

Grammars can be used to define a syntactic model of documents created in development. [Normark 89] provides a detailed discussion of programming environments that provide support based on syntax with a discussion of example systems. The Cornell Program Synthesizer [Reps 84] generated a syntax-directed editor based on the grammar of the programming language. Muir [Winograd 87] was a similar system that also included support for graphical editing of documents, and pattern-based transformations of programs[Normark 87]. In Muir, common editing tasks could be automated and different types of document views could be defined [Peyton 87]. The system was easily configured by adding or replacing syntactic models. Syntax-based approaches are limited, though, by the degree to which development work can be characterized by the grammar of the language used in implementing software.

A number of model-based environments have been built under the umbrella of the Knowledge-based Software Assistant project [Balzar et al 83]. Knowledge representation was used to model the different aspects of software development (programs, specifications, testing, project management etc.). Rules and templates were used to automate tasks performed in development. Usually, a formal specification would be defined which would be transformed into an executable program. The CHI system [Philips 83] was a self described programming environment which supported the definition of domain models, logic-based specifications in those domains, and pattern-based transformations to create programs from specifications. Gist [Johnson 87] took a similar approach but instead of logic-based specifications used natural language specifications. Programmer’s Apprentice [Rich 81] represented specifications in terms of a formalism called the plan calculus. A library of templates for common tasks called cliches was built that could be reused to automate programming tasks.   A limiting factor in knowledge-based approaches has been difficulty in encoding models, rules, and templates in a form which is formally correct, captures all the complexities needed for full industrial application, and is easily maintained by average developers.

Hypertext models have been used to create environments that provide support for ensuring the comprehensibility of software systems. The basic premise is that software documents are used to communicate information between participants as much as they are used to create a description that can be executed [Osterbye 93]. The HyperPro project [Osterbye & Normark 93] has created a system of nodes and links which can be typed in order to structure development on a software system. The system also supports the association of actions with nodes and links that can be used to accomplish tasks.

Work on object-oriented databases and repositories such as the Emerald project [Black et al 87] have laid a foundation for creating an environment based on object models. Distributed environments in which their may be many model-based servers and agents active have incorporated a standardized object-based communication protocol and/or broker such as described in [Stallings 93]. Integrated Project Support Environments (IPSEs) [Brown 91] are designed to support software development and project management in a flexible way that enables models for different aspects of works to be added and replaced.   This is related to work on Portable Common Tool Environments that provide a platform for defining of tools and models independent of a particular operating system or hardware configuration [Campbell 88].

Problem Solving Templates

Task templates are used in our architecture to characterize previously performed work in a manner that enables it to be reapplied in recurring tasks. Templates describe solutions to previously encountered problems. Our primary influence has been artificial intelligence work on automating problem solving, especially as it has been applied to automating software development.

Automatic use of templates, though, is not required for effective and dramatic improvements in the quality and productivity of knowledge work. Design patterns, such as have been identified in object-oriented design [Gamma et al 95] illustrate this point. Design patterns not only describe solutions to previously encountered problems, they identify the critical aspects of a problem in a manner which reflects a conceptual breakthrough in our understanding of the problem. A designer who applies a design pattern in their work, will create better software quicker.

An understanding of how a template can be integrated into the performance of a task is the next ingredient in effective use of templates. In our work on automating design, understanding the manner in which designers performed their work [Guindon 90] was critical for creating an environment in which there was an opportunity for templates to be applied. On-line documentation to organize a library of template descriptions, structured with effective search and browsing techniques [Nanard & Nanard 93] is one way to integrate the knowledge and experience reflected in templates into the domain of work.

Partial automation in the application of templates has been the goal in this thesis. We automate those aspects of template application which can be automated and providing flexible environment support to assist the participant with those aspects which can not be automated. Rules as used in expert systems [Davis et al 77] to characterize a single step in a task and can be viewed as a very simple template. One approach to partial automation is to define rules which are effective for a domain, and then provide a framework in which a participant can decide and select which rule to apply next. This approach has often been used in knowledge-based approaches to automating software development. Program transformations [Goldberg 86] are rules which describe how a specification document can be transformed to create a program document that is executable. The effects of rules can be characterized by constraints or logical conditions in order to automate their application. Preconditions determine when a rule can be applied and postconditions define the effects of a rule. Obligations are a special form of constraint [Perry 87] that we have used in our work to assist a participant in determining the sequence of actions to take.

Templates for software algorithms were called cliches in the Programmer's Apprentice project [Waters 86] and represented as plans. Partial automation was provided through an interaction in which participants could work either with program code or plan representations of their code. Cliches could be selected and modified by the participant to create a program, or the computer could be requested to implement the cliché or combine it with other cliches.

As we mentioned before, the power of templates is determined by the manner in which they identify the critical aspects of a problem in a manner to create a conceptual breakthrough in our understanding of a problem. We have seen this most visibly demonstrated in knowledge based approaches to software automation in the KIDS system [Smith 89]. The essential ingredients in creating algorithms which implemented specifications was identified as the application of general programming strategies such as divide and conquer, or global search. In the KIDS systems these strategies were characterized in a manner such that their application could be automated. One of the examples performed in the system was the generation of algorithms for sorting numbers. From a simple logical specification that a list of numbers should be sorted, KIDS generated several of the classical algorithms for sorting (quicksort, bubblesort) automatically depending on which strategy was applied.

1.5.2 Industrial Practice

The evolution of industrial practice over the last fifty years with respect to software automation has been driven by increasing complexity in the types of systems built. [Brooks 86] identified that the essential aspect of software automation is the fashioning of complex conceptual constructs. Increased capacity for software automation is achieved by increasing the granularity of the conceptual constructs we can use and reuse. Document models, tasks, and task templates in our architecture are mechanisms for increasing the capacity of software automation. The evolution of industrial practice is also driven by increasing complexity in the level of automation provided. Early efforts achieved a data level of interaction. Since then there has been a general migration to information levels and even knowledge levels of interaction.

Initially, the focus of software automation was in one's understanding of a programming language (primarily FORTRAN and COBOL). To provide a data level of interaction, one used a programming language to implement an algorithm that defined the computer's behavior in terms of data structures and actions. People interacted with software primarily through file I/0 to perform data processing and numerical calculations. The basic tools available to create software were text editors, code libraries, compilers and debuggers.

As the industry matured, the focus of software automation was captured in one's understanding of a methodology (primarily data modeling and object orientation). To provide an information level of interaction, one followed a methodological approach to building models of the domain and defining software systems in terms of those models. The methodology was a general one that was applicable across a wide variety of application domains. Software provided a communication infrastructure in the form of databases and networks, as well as tools such as word processors, data entry systems, and spreadsheet programs that could be used in one's work. Interactions with software were now made possible through graphical interfaces or communication interfaces between software systems. Development of software was supported by a variety of model-based case tools including diagrammers, code generators, class libraries and fourth generation language systems. IEW from Knowledgeware is a good example of a toolset which supported the Information Engineering methodology of James Martin. Smalltalk from ParkPlace was the first to introduce class libraries. PowerHouse from Cognos is a classic fourth generation language system.

Today, the focus of software automation is becoming one of understanding a specific application domain (a particular industry like banking or a particular type of work like customer support). To provide a knowledge level of interaction, one has to understand the terminology and operational practices that are fundamental to a particular domain. Software is now providing an infrastructure that automates communication and the management of communications such as database synchronization, work flow routing of electronic mail, and network management. It no longer provides tools to support work as much as it provides a medium for work which can be configured and automated by the person doing the work (office suites, group support systems, on-line publishing). Interoperability of software systems within a particular application domain is important in order to provide seamless automation of the work processes within a domain.

Today, the trend is to define standards which predefine for an entire industry the information models, business processes and architecture which should be used. Telecommunications Management Network [Aidarous & Plevyak 94] [CCITT 92a] is an example for automating the management of telecommunication networks. The CALS initiative [Walter 92] standardizes software that automates work processes involved in the management of documents in the defense industry. A number of initiatives are underway to standardize model-based interoperability of software systems : Corba , OpenDoc, and Ole [Orfali et al 96]. Graphical application builders, and libraries of software components which can be combined and configured are also providing tools for software development. Lotus Notes from Lotus, Visual Basic from Microsoft, Axiant from Cognos, and Delphi from Borland are good examples of such systems

It is the exchange of information that is being automated in these domains, and the work that is being automated can be characterized as knowledge work. We will return at the end of each chapter in this thesis to discuss the relationship between our architecture and the trend towards standardization.

2.0 Document Models

Our main thesis is that knowledge work can be automated by the embodiment of its documents and tasks in software. Our architecture for software automation embodies documents in the form of document models which define the information processing aspects of particular types of documents. In this chapter, we present a definition of the concepts used to build document models and discuss issues in implementing them.

There are a few points which should be remembered as we present our perspective on documents. We are interested solely in the information processing aspects of documents used in knowledge work which can be captured explicitly in a document model. Intuitively, a document is any standardized form of written communication, including such things as business reports, application forms, musical scores, screen plays, engineering drawings, etc. We leave the determination of what should be considered a document to the participants in a domain of knowledge. Once documents have been identified, the level of software automation that can be provided is determined by the richness of the document models that can be defined and standardized The transition from paper documents to electronic documents also brings with it new possibilities that changes our understanding of documents. In particular, documents in an electronic form need not be passive or self-contained. Our approach to document models will include definitions of behavior and definitions of associations between documents.

2.1 Electronic Documents

"... a missionary ... had written a note to his wife on a wooden chip and asked a Fijian chief to deliver the message. The chief was scornful of the errand and asked, 'What must I say?' ... 'You have nothing to say', the missionary replied. 'The chip will say all I wish.' With a look of astonishment and contempt, the chief held up the piece of wood and said, 'How can this speak? Has this a mouth?'"

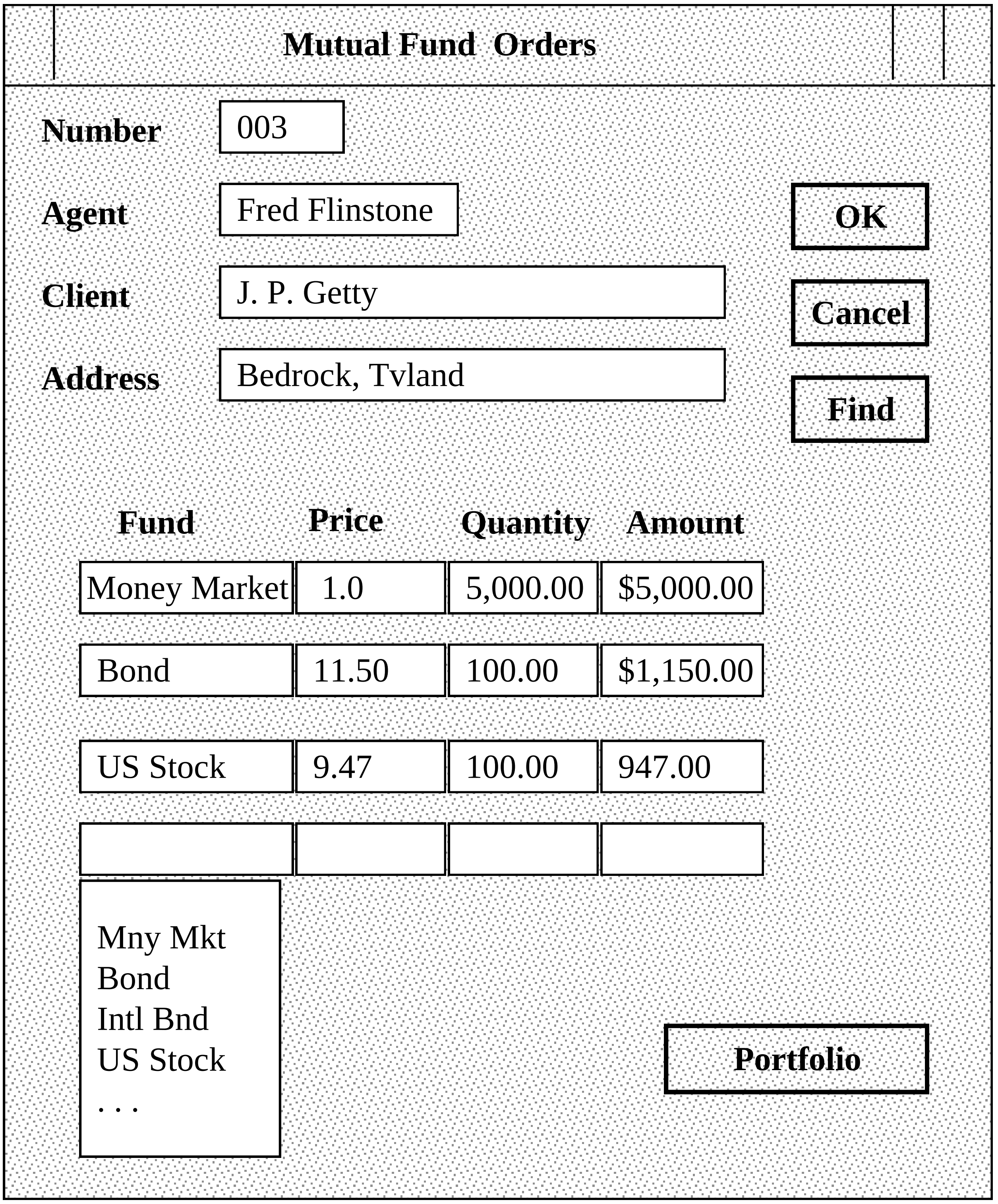
J.R. Clammer, Literacy and Social Change: A Case Study of Fiji (Leiden, The Netherlands: E.J. Bulle 1976), p. 67

2.1.1 The Vision

It is a premise of this thesis that knowledge work is centered around documents. Documents are a form of communication, which we define as *the exchange of information between individuals through a common system of symbols, signs, and behaviors for the purpose of conveying knowledge*. An electronic document, which we define as *the embodiment of a document in software*, is a basis for the automation of work because it creates a system that automates the exchange of information (for the purpose of conveying knowledge) on behalf of individuals.

The documents used in knowledge work can be categorized into types (order form, product catalog, software design ...). Each type of document groups and relates particular types of information differently in order to convey different types of knowledge. An electronic document is the embodiment of a single type of document[2]. To create an electronic document, the document type must be modeled as a system of information exchange. The types of information that it organizes must be in the form of electronic data. Actions must be defined to enable the exchange of information. Once defined, the model serves as a blueprint for the implementation of the electronic document.

[2] A document agent (discussed in Chapter 3) provides an interface through which one can interact with one or more electronic documents


Mutual Fund Orders

Number	003
Agent	Fred Flinstone
Client	J. P. Getty
Address	Bedrock, Tvland

Fund	**Price**	**Quantity**	**Amount**
Money Market	1.0	5,000.00	$5,000.00
Bond	11.50	100.00	$1,150.00
US Stock	9.47	100.00	947.00

Mny Mkt
Bond
Intl Bnd
US Stock
. . .

Figure 7. Form Document

2.1.2 Scenario

Let us consider a scenario which will illustrate an electronic document. In figure 7, is a picture of a Form document that is used to take purchase orders from clients who wish to invest in mutual funds. There are text fields for the order Number, the purchasing Agent, the Client, and for each of the mutual funds purchased there is the Fund, Price, Quantity and Amount for each purchase. There are also buttons which allow the user to enact the order (OK), cancel the order (Cancel), find an existing order (Find) and display another Form document to show the client's current Portfolio. Special behavior is associated with some of the fields. When entering the Fund, there is a select list of available funds from a product catalog. The Price for the fund is entered automatically, once a fund is selected. The Amount is calculated automatically, once the Quantity has been entered.

If we view this form document as a system of information exchange, some observations can be made. The document consists of *parts*. There are fields and buttons. The documents and the parts convey *information*. They have labels which describe their use and values which have meaning. The documents and its parts have *behavior*. Fields can be selected and edited, while pressing buttons causes information to be processed and other documents to be displayed. Finally, there are *associations* between parts and with other documents. The value in the Amount field is determined by the Price and Quantity field. One can access a client's portfolio Form from this order Form and presumably the order form is accessible from other documents like a client Form.

Digging deeper, we realize that, in addition to its visible parts, the form document must have special "query" parts which organize the information associated with an Order and a Catalog that other parts of the form access. Order contains all the information relevant to a purchase order. Catalog contains all the relevant information about products that can be ordered. The information in Order is accessed by fields in the form document for the purposes of display and modification. There is an association between the information in an Order and the information in the Catalog which is used to fill the select list of the Fund field, and which is used to fill the Price field once a fund is selected. Order and Catalog are also associated with Table documents in a database where their information is stored and retrieved. This behavior of Order and Catalog is triggered by buttons in the Form Document.

The Form document for taking purchase orders is a system of information exchange. It automates the exchange of information necessary for making a purchase. It can also automate the exchange of information between it and related documents that are used in the organization's business process for carrying out that purchase order.

2.2 A System of Information Exchange

In this section, we will provide a rigorous definition of the concepts which are used to build document models. Our intention is to model the information processing aspects of electronic documents which enable information to be communicated. In doing so, we choose to view a document as a system.

The Delta project [Haanlykken & Nygaard 81] defined systems in general as follows:

> "A *system* is a part of the world which a person or a group of persons during some time and for some reasons, choose to regard as a whole consisting of parts called *components*. Each component is characterized by selecting properties, and the actions which may involve itself and other components."

In work on the Aleph system specification language [Osterbye 89] added that:

> "It is also considered important that one can express *relations* between the components, as well as putting global *constraints* on the systems behavior."

Our interpretation of documents as a system of information exchange is illustrated in Figure 8. There are two instances of electronic documents shown which contain objects that group information within the documents. Data values in each object characterize the information contained in the documents at a particular moment . Events can occur which exchange information between objects. There are also links between objects that capture the associations between different groups of information in the document instances. The diagram indicates that information in different electronic documents can be linked, and there are events which can occur that exchange information between different documents.

A document model will define document instances in this manner using the concepts of component, data item, action and relationship to define the objects, values, events, and links which make up a document instance. In the rest of the section these concepts are characterized precisely and then illustrated with examples from our scenario.

We have chosen not to use a precise language or formal notation in which to express document models, although our presentation follows a notation loosely based on typical approaches to defining class concepts and their properties. We have wanted to make clear, at a high level, the essential aspects of document models without tying them to a particular framework of expression. When document models are implemented in software, the choice of language will necessitate that some aspects of document models be compromised. We will discuss the language issues involved in section 2.3.

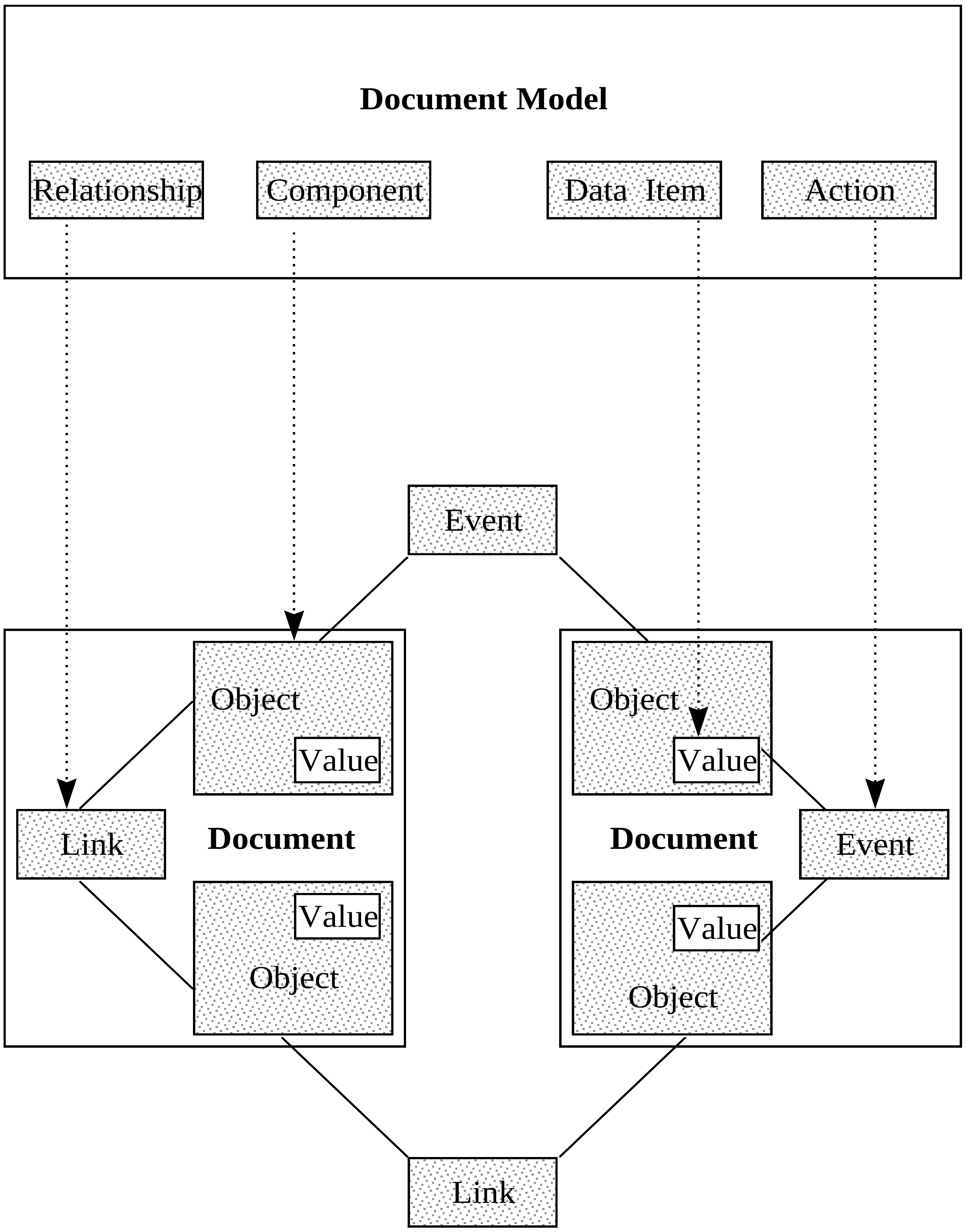


Figure 8. Understanding Documents

2.2.1 Document Model

Document Model:	A representation of a system of information exchange in terms of its parts, information, behavior and associations.
Document:	A system of information exchange. An instance of a document model.

A document model represents a system of information exchange that characterizes a type of document. In our architecture, documents will be created as instances of a document model. As such, documents will be typed and are defined by exactly one document model. For example, when we say that the purchase order form in our scenario is a Form document, we mean that the order form is a document of type Form. Its parts, information, behavior and associations are defined by the Form document model.

Properties

Name:	A document model has a name which is used to identify the model, and which is used as the type for documents that are instances of the model. A document has a name which, together with its type is used to identify the document.
Parts:	A document model has parts. Each part is defined by a *component*. The parts of a document are *objects* which are instances of the components defined in its document model. The document is considered the owner of its parts. Only the document can add or remove one of its parts. If a document is deleted, so are its parts.
Information:	A document model has information. Each unit of information is defined by a *data item*. The information in a document consists of *values* which are instances of the data items defined in its document model.
Behavior:	A document model has behavior. Each unit of behavior is defined by an *action*. The behavior of a document consists of *events* which are instances of the actions defined in its document model.
Associations:	A document model has associations. Each association is defined by a *relationship*. The associations of a document are *links* (*to* and *from* other documents or objects) which are instances of the relationships defined in its document model. Links with other documents or objects in other documents are called *external* links. We will distinguish between *owned relationships* and *attached relationships*. Owned relationships define the associations of the document model. These correspond to "to links". Only the document can add or remove them. Attached relationships can be added to the document by any other document or object. These correspond to "from links". Only the document or object which adds an attached relationship can remove it. Attached relationships are owned relationships in the document model or component of the document or object which adds it.

Example

Using this definition, we can create a document model for the Form document which we used for our scenario. Our Form has field(s), button(s), and query(s) as parts. These group most of the information, behavior, and associations contained in the document. The only additional information that the Form contains is a title. Its behavior consists of the ability to display itself to the user. The Form also initializes that display when it is first created (and when it is reset by the Cancel button). The Form can be associated with form(s) which are called by it. It can also be associated with form(s) which call it.

Form **Document Model**

Parts:		Field[n], Button[n], Query[n]
Information:		Title[1]
Behavior:		Display, Initialize
Associations:		
	Owned:	Calls (Form, Form)[n]
	Attached:	Calls (Form, Form)[n]

Using the document model we have created, we can describe the Order form shown in Figure 7. Each of the fields, buttons, and queries in the order form are objects. The title of the form is a string. The behavior of the form is captured by events as they occur. For example, the form is displayed, its fields are initialized, other events occur which are associated with its fields and buttons, and then a Portfolio form is displayed. The Order form is linked to the Portfolio form it called, and it has a link from the Client form which called it.

Order **Form**

Objects:		Number, Agent, Client, ... Fund1, Quantity1 ... and Amount4 **Fields**
		OK, Cancel, Find, Portfolio **Buttons**
		Order, Catalog **Querys**
Values:		**Title** “Mutual Fund Orders”
Events:		**Display**(Order **Form**), **Initialize**(Order **Form**), ... **Display**(Portfolio **Form**)
Links:		
	To:	Portfolio **Form**
	From:	Client **Form**

2.2.2 Component

Component: A representation of a subsystem of information exchange in terms of its parts, information, behavior and associations.

Object: A subsystem of information exchange. An instance of a component that is contained in a document.

A component represents a subsystem of information exchange that characterizes a type of part within a document. Objects are created as instances of a component. As such, objects are typed and are defined by exactly one component. For example, when we say that the OK button in our scenario is a Button, we mean that the OK button is an object of type Button. Its structure and behavior are defined by the Button component.

Properties

Name:	A component has a name which is used both to identify the component and as the type for objects which are instances of the component. An object has a name which, together with its type, is used to identify the object.
Parts:	A component has parts. Each part is defined by a *component.* The parts of an object are *objects* which are instances of the components defined in its component. The property of parts is a recursive one that organizes all the subobjects of an object into a hierarchy. An object is considered the owner of its parts. Only an object can add or remove one of its parts. If an object is deleted, so are its parts (and therefore the entire hierarchy of subobjects).
Owner:	There is exactly one *object* or *document* which an object is part of. That object or document owns the object. The owner is defined by the *document model* or *component* of the document or object which owns it. A component can be specified as a part in several document models or components. This increases the reusability of components.[see American Programmer for a discussion of issues related to components]
Information:	A component has information. Each unit of information is defined by a *data item*. The information in an object consists of *values* defined by the data items of its component.
Behavior:	A component has behavior. Each unit of behavior is defined by an *action*. The behavior of an object consists of *events* defined by the actions of its component.
Associations:	A component has associations. Each association is defined by a *relationship*. The associations of an object are *links* (*to* and *from* other documents or objects) which are defined by the relationships of its component. Links with other documents or objects in other documents are called *external* links. We will distinguish between *owned relationships* and *attached relationships*. Owned relationships define the associations of the component. Only the component can add or remove them. Attached relationships can be added to the component by any other document or object. Only the document or object which adds an attached relationship can remove it. Attached relationships are owned relationships in the document model or component of the document or object which adds it.

Example

Here is a component definition for the fields that are part of the Form document model. The parts of a field are Procedure(s) and a SelectList. The procedures determine how the field behaves. They are parts, not behavior, because the procedure part chosen allows one to customize the behaviors defined in general for fields (as we will see with the instance). The SelectList is an optional part that provides a list of values to choose from. The information for the field consists of the Entry it is currently displaying, a DefaultEntry that is displays initially or when it is reset, and a QueryItem which is the name used to associate this entry with information in a Query used by the Form. The behavior of the field is defined by actions which Initialize the field, can get or set the Entry of the field, get a SelectList to choose from, and update the associated Query with the new value of its Entry. The Field owns relationships which link it to other fields that it sets entries for, and to the concept which it edits. There is also a linked relationship from fields which set it.

Field **Component**

Parts:		Procedure[n], SelectList[1]
Information:		Entry[1], DefaultEntry[1], QueryItem[1]
Behavior:		Initialize, GetEntry, GetSelectList, SetEntry, UpdateQuery,
Associations:		
	Owned:	Sets (Field, Field)[n]
		Edits (Field, Query) [1]
	Attached:	Sets (Field, Field)[n]

Here we have the Fund1 Field from Order Form. It has a part Select1 that provides a SelectList of values to choose from. It also has a SetEntry Procedure part that customizes the behavior of the SetEntry event. SetEntry checks to see if Fund1 has a SetEntry Procedure it calls an Execute action which in this case not only sets Fund1's entry, but it also sets the entry for Price1 as well. Currently, the Entry for Fund1 is set to "Money Market" and it is editing the FundName value of the Order Query.

Fund1 **Field**

Objects:		SetEntry **Procedure**, Select1 **SelectList**
Values:		**Entry** "Money Market", **DefaultEntry** " ", **QueryItem** "FundName"
Events:		**Initialize**(Fund1 **Field**), **GetSelectList**(Fund1 **Field**), **SetEntry**(Fund1 **Field**,"Money Mrket"), **Execute** (SetEntry Procedure), **SetEntry** (Price1 **Field**, 1.0) ...
Links:		
	To:	Order **Query**, Price1 **Field**
	From:	

2.2.3 Data Item

Data Item:	A representation of a unit of information.
Value:	A unit of information. An instance of a data item that resides in a document or object.

Data items represents a unit of information that is grouped within a document model or component. Values are created as instances of a data item. As such, values are typed and defined by exactly one data item. For example, when we say that "Mutual Fund Orders" is the title of the Order Form, we mean that is a value of type Title as defined in the Form document model.

Properties

Location:	A data item has a location, which is the document model or component in which it resides. A data item has exactly one location. Values defined by a data item have a location which is a document or object that is the instance of the document model or component in which the data item resides. A value has exactly one location. Values are only accessible by the object or document in which it resides.
Name:	A data item has a name which is used both to identify the data item and as the type for values which are instances of the data items. Values do not intrinsically have a name. If there is only one instance of a value in an object or document, then it is uniquely identified by its type and the name of the document or object it resides in. If there is more than one instance of a value, the object or document may have mechanisms, defined in its component or document model, for associating labels with values in order to access them effectively. Alternatively, it may have mechanisms for finding values that match a pattern.
Pattern:	Data items have a pattern which describes the range of possible values for its instances and the information exchanges which can be applied to them. This pattern determines how data items can be accessed by actions. In the simplest case, a pattern would simply be a data type declaration in some language.

Example

The Form document model we created has a Data Item named Title. The pattern used to define Title is simply a reference to a predefined type "STRING" of size 32. This means that Title will be a fixed length string of size 32, and that behavior of the Form document model will be able to define information exchanges involving Title with the usual operations associated with strings (assignment, concatenation ...).

Title **Data Item**

Location:	Form
Pattern:	STRING[20]

In the Order form document we created, there is a value "Mutual Fund Orders   " of type Title. Initially, when the Order form document was instantiated it would have had a value " " (empty string) of type Title. At some point, the event SetTitle occurred in Order form, that resulted in that value being exchanged for "Mutual Fund Orders".

"Mutual Fund Orders" **Title**

Location:	Order **Form**

2.2.4 Action

Action:	A representation of an exchange of information of a component or document model.
Event:	An exchange of information of an object or document. An instance of an action.

Actions represents an exchange of information that can be performed by a document model or component. An event is created as an instance of an action. As such, events are typed and defined by exactly one action. For example, when we say that SetEntry(Fund1,"Mny Mkt") is an event performed by field Fund1, we mean that it is an event of type SetEntry as defined in the Field Component.

Properties

Location:	An action has a location, which is the document model or component in which it resides. An action has exactly one location. Events defined by an action have a location which is a document or object that is the instance of the document model or component in which the action resides. An event has exactly one location.
Trigger:	An event has a trigger, which is the request that causes an event to occur. This request is made from an event performed by a document or component or outside agent[3] that has access to the object where the triggered event resides. A document or object has access to an object if it owns the object or is linked to the object.
Name:	An action has a name which is used both to identify the action and as the type for events which are instances of the action. Events do not intrinsically have a name. If there is more than one instance of an action in existence, an event can be uniquely identified by its trigger. Events are created when they are triggered and deleted when the information exchange defined by their action is complete.
Inputs:	An action has inputs, which define the information that must be provided by the trigger of an event. An input can be a document model, a component, or a data item of the component where the action resides. A trigger provides documents, objects or values as inputs to an event.
Outputs:	An action has outputs, which defines the information that will be provided to the trigger of an event in exchange for inputs. An output can be a document model, a component, or a data item of the component where the action resides. An event provides documents, objects or values as outputs to its trigger.
Pattern:	Actions have a pattern which describes the exchange of information which takes place in an event. A pattern can describe events involving inputs and outputs, and the parts, information, behavior and associations of the component or document model where the action resides. The following events are *primitive*: getting or setting values, creating or deleting objects, creating or deleting links. Any other event can be defined as a *compound event* which triggers a collection of such events and other compound events. A compound event can trigger events which reside in the object where the compound event resides, or in the objects linked to or owned by that object. Patterns are written in a language which implicitly or explicitly describes primitive events and the creation of compound events from them.

[3] Document agents are discussed in Chapter 3. All behaviour associated with a document originates with a document agent.

Example

Here we show the definition of the SetEntry action for a field. The SetEntry action changes the Entry data item of field, which is used to store the entry that a field on a form displays. It takes as input an Entry data item that will be exchanged for the Entry data item of field. The new value of the field's Entry data item will be returned as the output. Depending on how the SetEntry action is defined, it is not necessarily set to the value of the input Entry. If the input Entry is invalid it might be ignored, or there may be a special computation associated with this field (it might automatically converts strings to upper case for example).

The pattern that describes the action is written in a weakly typed procedural language. It checks to see if there is a procedure object called SetEntry that is part of the field. If there is special processing associated with SetEntry for this field then it will be defined in the SetEntry procedure object. If the procedure object exists, then it is executed. If it is not defined, then the SetEntry data item is changed to the value of the Input SetEntry. The new value of the field's SetEntry data item is returned as an output.

SetEntry	**Action**
Location:	Field
Inputs:	Entry
Outputs:	Entry

Pattern:

```
Action Field.SetEntry (Input Invalue Entry; Output Outvalue Entry;)
    Proc = Field.GetPart("Procedure", "SetEntry")
    IF (Proc)
        THEN  Proc.Execute(Invalue,Outvalue)
        ELSE  Field.SetValue("Entry", Invalue)
    ENDIF
    Outvalue = Field.GetValue("Entry")
    End Action
```

Here we have an event defined by the action above. The event has been performed by field Price1. It was triggered by the setting of field Fund1's entry to "Mny Mkt". Presumably there is a SetEntry procedure that is part of field Fund1. It specified special processing that caused the Price associated with Mny Mkt to be set in field Price1 automatically. The input to the event was a price of 1.0. Which is the new value of entry for Field Price1 returned as an output from this event.

SetEntry	**Event**
Location:	**Field** Price1
Trigger:	SetEntry (**Field** Fund1,"Mny Mkt")
Inputs:	1.0
Outputs:	1.0

2.2.5 Relationship

Relationship: A representation of a connection from an object or document to other objects or documents that makes them accessible to each other.

Link: A connection from an object or document to other objects or documents. An instance of a relationship.

A relationship is a representation of accessibility between components and/or documents that is significant to the knowledge conveyed by the document. A link is created as an instance of a relationship. As such, links are typed and defined by exactly one relationship. For example, when we say that there is a Calls link from the Order Form to the Portfolio Form, we mean that it is a link of type Calls as defined by the Calls relationship in the Form document model.

Properties

Location: A relationship has a location, which is the document model or component in which it resides. A relationship has exactly one location. Links defined by a relationship have a location which is a document or object that is the instance of the document model or component in which the relationship resides. A link has exactly one location. Links are owned by the object or document in which it resides. Only the owner of a link can create or delete it.

Attachments: A relationship has attachments which are the document models or components that are connected to the owner of the relationship. A link has attachments that are documents or objects which are instances of the attachments of the relationship that defines the link. The attachments of a link can not create or delete it, but they can access the link to determine the owner of the link. Links with attachments that are other documents or objects in other documents are called *external links*. External links are a mechanism for exchanging information in other documents. The owner of a link can create events that trigger events in the attachments of a link, including external links.

Name: A relationship has a name which is used both to identify the relationship and as the type for links which are instances of the relationship. Links do not intrinsically have a name. If there is only one instance of a link in an object or document, then it is uniquely identified by its type and the name of the document or object it resides in. If there is more than one instance of a link, the object or document may have mechanisms, defined in its component or document model, for associating labels with links in order to access them effectively. Alternatively, it may have mechanisms for finding links that match a pattern.

Pattern: Relationships have a pattern which describes the range of possible attachments for its owner. Links created as instances of a relationship will conform to that pattern.

Example

The Form document model we created had a relationship Calls which was used to associate forms when there was a call from one form to another. Here we show the relationship that defines that connection. It is located in the form document. The relationship can be attached to form(s). Its pattern indicates that it is intended to capture an association in which an event in one form triggers an event in another form. This is a useful relationship for maintaining the integrity of work involving several forms. If the called form changes, it could affect the calling form (especially if the change eliminates the event which was triggered in the called form).

Calls **Relationship**

Location: Form

Attachments: Form[n]

Pattern:

```
Relationship(Calls)
        Exists Event E1, E2, such that
        Location(Calls) = Location(E1) &
        Location (E1) in Attachments(Calls) &
        Triggers (E1, E2)
```

We see that in the particular case of the Order form. There is only one form that it calls. However, we can also see that there are two forms which call it: the Client form where basic information about the client is located, and the unfilled orders form which lists orders that have not been filled yet.

Calls **Link**

Location Order **Form**

Attachments Portfolio **Form**

Calls **Link**

Location Client Form

Attachments Order Form, Proposal Form

Calls **Link**

Location Unfilled Orders Form

Attachments Order Form, Client Form

2.3 Language Issues

We have defined a document model as a system of information exchange. Our intention when creating a definition of a document model in this manner, is that it will be implemented in some programming language or system. A document model implemented in this fashion will simulate the behavior of documents (of the type defined by the model) and their objects, values, events, and links.

There are a number of issues which arise in implementing a document model as we have defined it. The mechanisms for defining the aggregation of components into parts, the aggregation of data items into information, and the aggregations of links into associations will depend on the choice of programming language or system to use. Each type of language or system will force a different type of compromise.

The mechanisms for defining patterns in data items, actions and relationships will also depend on the mechanisms available within the language or system used. The data types available determine what sort of information can be captured in a document (text, numbers, or even sounds and pictures). The mechanisms available for defining patterns determine not only how information is organized but also how it can be processed within a document.

2.3.1 Aggregation

For each unit of our aggregations, it is important to consider the cardinality, optionality and choice allowed. In our presentations, we have indicated the cardinality of units by a number within square brackets, so that Field(n) indicated there could be any number of Fields as Parts of a Form, while Title(1) indicated that there was exactly one Title in the information for a Form. The optionality of a unit is also important. When we said Title(1), it could have meant at most one or exactly one. Choice is relevant as well. We said that a field had an optional part SelectList. It is useful to be able to say that a field has a part which could be a simple edit box or a SelectList but not a multi-valued radio button.

There are different mechanisms for identifying a unit within an aggregation. Parts could be organized as a random list, an array, a heap or any number of complex data structures. Labels might also be attached to parts. We specified Entry and DefaultEntry as two different data items in Field, but in reality they are both the same data item used for different purposes. Another approach. Where units are not labeled, search mechanisms would be appropriate. Searches find units, based on a pattern or filter.

Different approaches can be taken with respect to these issues. A grammar-based approach captures the decomposition of a document into its parts as a syntax defined by production rules. This is effective to model the parts and choice of parts in a document, especially when data will be primarily in the form of text. However, special mechanisms must be built for a grammar-based approach to capture actions associated with components, relationships between components, or to use complex data structures for aggregations.

A data-based approach captures components as rows of data in a table and captures relationships as relations between tables. This is effective to model the grouping of information and associations between information in a document. However, special mechanisms must be built for a data-based approach to define the decomposition of a document into its component parts, support choices of different types of components, capture actions associated with components, and use complex data structures for aggregations.

An object-based approach captures components as object classes, data items and relationships as attributes of those classes, and actions as methods. This is effective to model the grouping of information and associations between information in a document in a manner that integrates behavior. Inheritance can be used to enable choice. However, special mechanisms must be built for an object-based approach to distinguish the parts hierarchy of components in a document, and maintain the attributes which capture relationships. Direct support for document models (though less efficient) could be achieved by implementing each of the concepts we have identified (document models, components, data items, actions, and relationships) as an object class.

2.3.2 Patterns

The mechanisms in a language for specifying patterns are often different for each of data items, actions, and relationships. To enable customization which improves automation, an important feature is the ability to extend or modify a pattern dynamically after a document model has been implemented.

Data items are typically specified by data types, but that can be extended to include default values, and restrictions on values in the form of validations. Support for defaults and validations can also be implemented in get and set actions which access values. In some cases, it may be more appropriate not to maintain a value for a data item, but simply compute it in a get action. One mechanisms for extending or modifying data items dynamically is to allow restrictions or extensions to the range of values. For example, in an office providing support in a local area, one might want to restrict the range of values for postal code to only those codes applicable in the local area. This is easily supported if the range of possible values for a data item is listed explicitly. It could also be supported by allowing dynamic modifications to the get and set actions which access a data item.

Actions are typically specified by procedural code. Document models, should normally restrict themselves to defining a basic set of primitive actions that will be required independent of the particular application domain or task which might use them. Domain or task specific behavior can then be created by combining or customizing primitive actions. In many cases, greater customization is enabled when an action is declared in terms of its inputs and outputs, but no pattern is specified. The implementation must then support a mechanisms for specifying behavior dynamically. This is usually provided in the form of a scripting language which can be interpreted or compiled dynamically. Actions which have been predefined can be customized if pre and post actions can be specified dynamically. When an event is triggered, the behavior specified by the pre action occurs, then the behavior which was predefined for the action occurs, and then the behavior specified by the post action occurs.

Relationships are typically specified as a combination of data type and constraints. It is not just the types of components that make up a relationship that are significant, but also the meaning associated with that relationship. That meaning is captured by constraints which specify the properties about the owner and attachments of a relationship which should be present. There may also be special processing associated with a relationship to ensure the validity of constraints associated with it. The integration of relationships with procedural actions which can affect the validity of constraints can be complex to manage. Often times, relationships are used to indicate constraints which should hold, but might not. Mechanisms are then implemented to notice and notify when actions have invalidated the constraints associated with relationships.

2.4 Project Examples

In this section we will illustrate our theory concerning document models with examples of actual models used in our project work. In the SoDA system (introduced in section 1.4.1), a single document model was used to capture a design representation of a software system. The model supported diagramming interfaces that provided a data flow and a control flow view of a software system. Relationships and constraints were used to capture and maintain the dependencies between different modules of the software system. We discuss some of the issues which arose in implementing relationships and constraints.

In the Axiant Prototype (introduced in section 1.4.2), several document models were used to represent the various types of program objects used in building a database system. The models supported a variety of graphical tools and editors which were used to create and maintain database systems. As part of the project, a diagrammer and code generator were built to support the definition of document models and the generation of code to implement them. That implementation included support for shared network access to documents defined by document models. We discuss some of the issues which arose in supporting inheritance, complex aggregation and relationships between document models.

2.4.1 System Document Model for Design Diagrams

The SoDA system automated software design work and was applied to the task of designing expert systems that were built as variations on a Blackboard architecture. An expert system is a software system that takes domain specific knowledge encoded into rules (usually of the form IF..THEN) that are stored in a knowledge base, and takes a general inference engine or interpreter, to apply those rules to some problem solving task in the domain. There are a number of different variations and optional components in a Blackboard architecture which might be used depending on the nature of the problem being solved, the types of rules used, and the type of reasoning applied in solving the task.

In Figure 9, a dataflow view of a sample blackboard system is shown at different levels of abstraction. At the most abstract level, a blackboard system consists of a knowledge component, where the rules and inference engine are located, and a blackboard where the problem description and solution are located. The knowledge component is triggered by events that occur on the Blackboard (SelectedEvent) and makes modifications to the Blackboard (BBmods) as a solution is built.

At the next level of abstraction, the Knowledge Component consists of a Knowledge Base, in which rules are grouped into Knowledge Sources relevant to different parts of the problem, and a Control which determines in what order Knowledge Sources should be applied to the problem, and a KSInference engine which applies the rules of a Knowledge Source to the problem. The Control takes Knowledge Sources (ApplicableKSs) which match the current state of the Blackboard (SelectedEvent). It selects one Knowledge Source (Ksi) and passes it to KSInference with an indication of what part of the problem to work on (Focus). KSInference makes modifications to the Blackboard (BBmods) based on the rules in the knowledge source.

In SoDA, the software system under design was treated as a single document. The different diagramming interfaces provided views of that underlying document. The interfaces could also be implemented using document models. However, our emphasis in SoDA, and in our presentation here, is to build a document model that defines the actual software system. The nesting of components within components in the diagram is used to show the hierarchy of parts in our Blackboard System. The links between components in the data flow view will be represented by an Input / Output relationship in our document model.

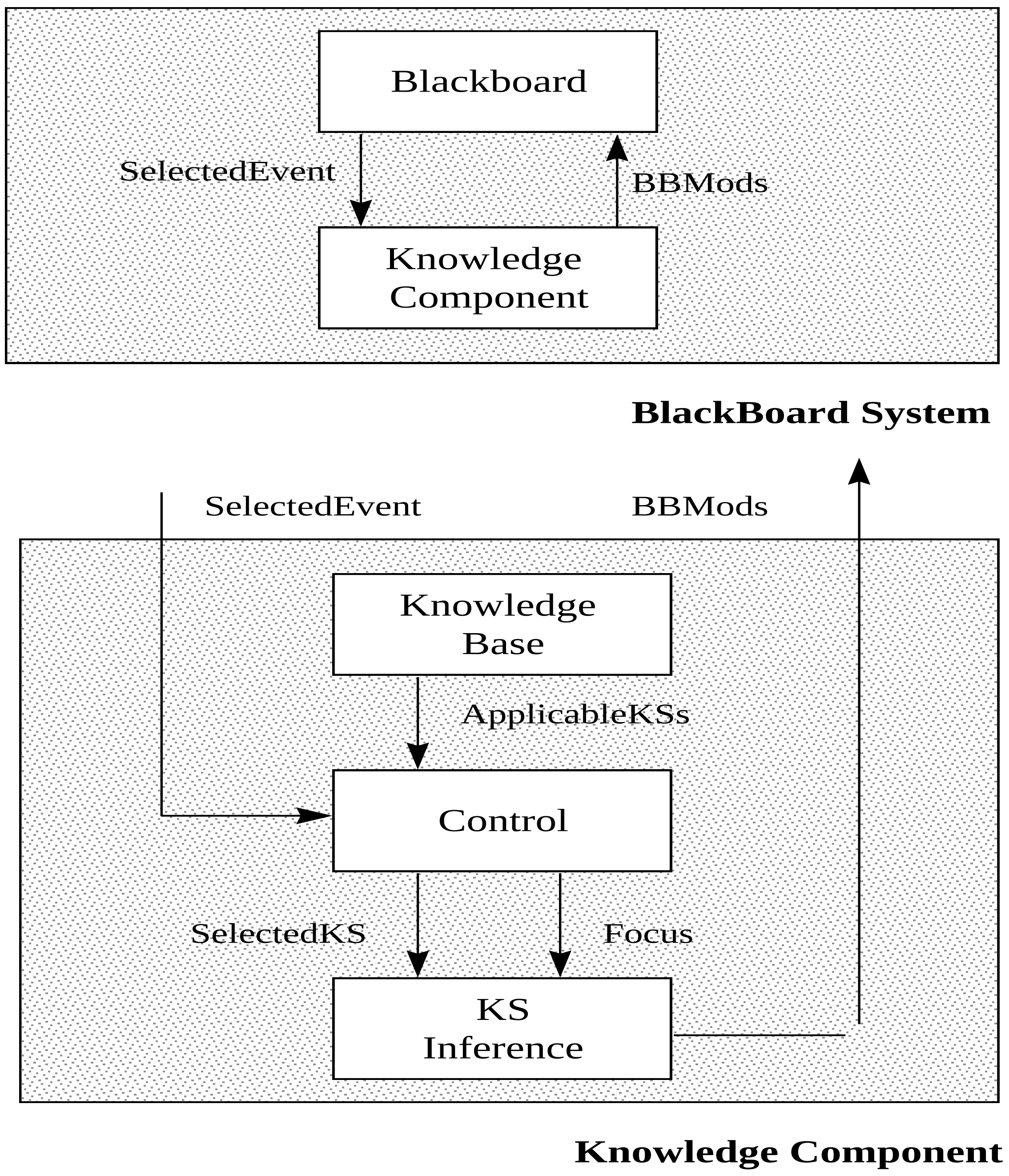


Figure 9. Data Flow Diagrams

In Figure 10, we have a control flow view of the same Blackboard System. The Control is activated by the BlackBoard System at start up. While there is a Focus, SelectedKS and Focus are passed to KSInference, which will create and execute a closure (a KS with an environment in which to execute). Then, SelectFocus is activated to select the next event to look at and create a focus based on that event. The focus is then passed to a Scheduler which determines which Knowledge Sources match the preconditions and selects one.

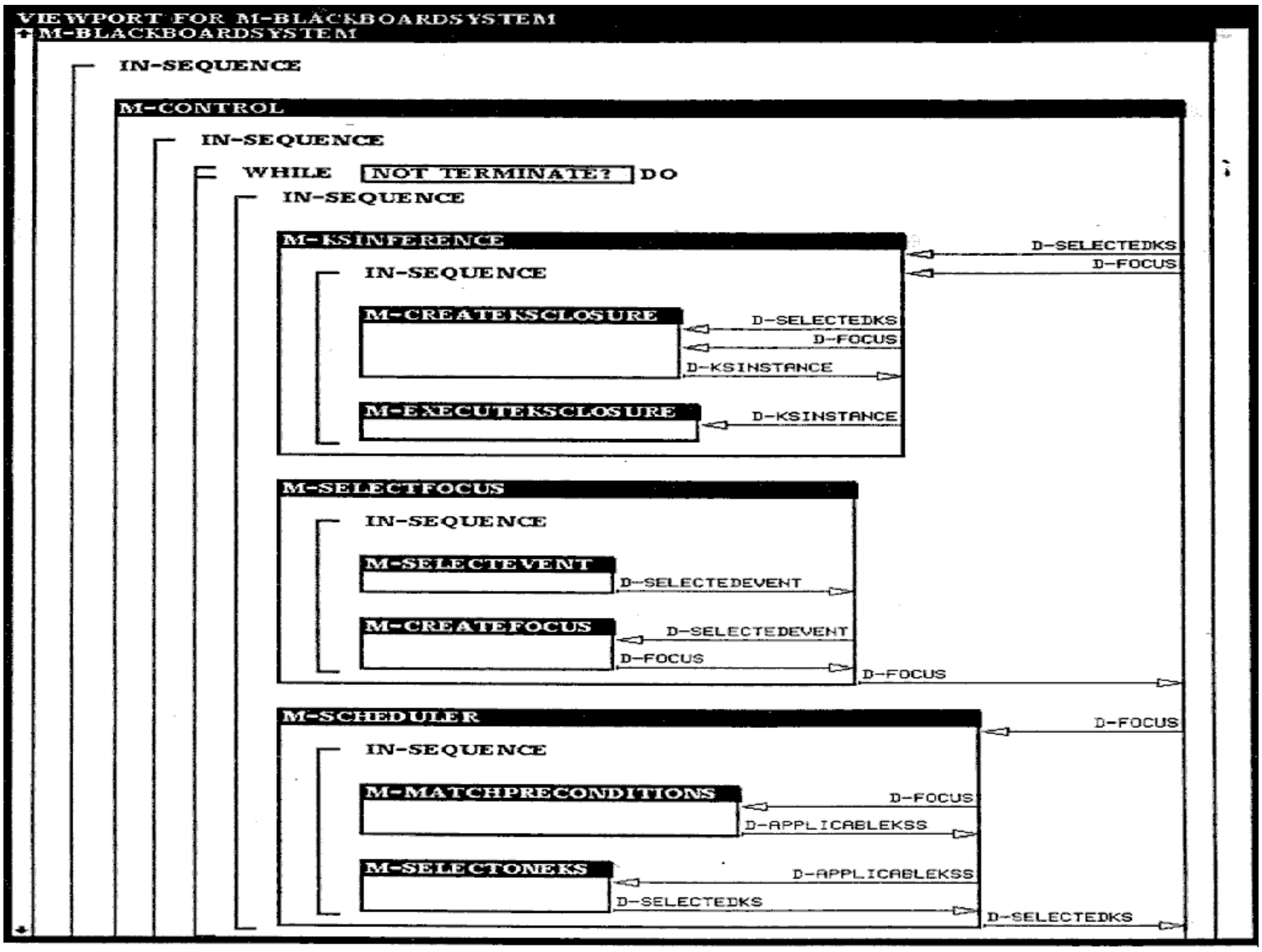


Figure 10. Control Flow Diagram

The nesting in the diagram is used to show the chain of links defined by a Calls relationship. The layout of the boxes on the page is also important. Components at the top of the page execute first starting with the outermost box. There are extra labels which indicate loops, versus sequences of operations. The arrows indicate dataflow between the calling component and the called component. Not surprising the control flow does not necessarily follow the static embedding of functions indicated by the part relationship. SelectEvent, for example, is part of the Blackboard component.

The System document model underlying these different views is quite simple. There are two components: Module and Data which are parts of System. Data has no parts, while Module has parts which are themselves Modules. Module is intended to represent the different software modules within a system, while Data models the information which flows between them. One of the principle tasks in design work is to ensure the completeness and consistency of a system by focusing on its decomposition into modules and the interactions between those modules. Our System document model is intended to be used to build an electronic document that automates the information exchanges underlying that task.

System **Document Model**

Parts: Module[n], Data[n]
Information:
Behavior: Get/Add/RemoveParts

Data **Component**
Parts:
Information: DataType
Behavior: Get/SetInformation, Get/Add/RemoveRelationships
Relationships: Source(Data, Module,)[1]

Module **Component**
Parts: Module[n]
Information: ActivationRule [1]
Behavior: Get/SetInformation, Get/Add/RemoveParts, Get/Add/RemoveRelationships
Relationships: Calls(Module,Module)[n], Inputs(Module,Data)[n], Outputs(Module,Data)[n]

The System documents created based on this model were intended to be designs of systems that would be implemented in Lisp. Each Module would become a Lisp function and each Data would become a argument or return value declaration in each function that used it[4]. The ActivationRule data item was a description of the processing within a Lisp function. Initially, this was written in a syntax that supported action diagrams. The ActivationRule could be parsed to determine the Calls, Inputs, or Outputs relationships or one could define Calls, Inputs, or Outputs links and then write an ActivationRule. Later, when the system was implemented, an external link to the file containing the source code was added.

Calls were relationships between modules, while Inputs and Outputs were relationships between modules and data. A Data component, tracked a piece of information important to the system. The DataType data item defined the type of information. For a particular Data, the Inputs and Outputs relationships could be used to trace the flow of data from module to module. Each Data identified a single Source which provided the information to other modules. Those modules might pass the data on to other modules, but each piece of information in the system would have a single source.

The only behavior defined in the system model were the primitive actions necessary for the get/set of information, and the get/add/remove of parts and relationships. The display behavior, and any complex edit operations, needed in the diagram interfaces were defined as part of the diagram interface with calls to the primitive actions in the system model.

The part relationship is captured by the Submodules and Supermodules attributes. While the semantic relationships for Control Flow and DataFlow are captured by Calls/Calledby and Input/Output. The Activation Rule specified the processing done by the module using an action diagram notation. In essence, the documents defined the interfaces between Modules.

[4] The System model left many details as implementation decisions. If a Data was input to the Module, that meant either it was passed in as an argument when the component was called, or it was the return value from a Module called by that component. Similarly, if a Data was output from a module, that meant, either it was return value from the component when it was called, or it was passed as an argument to a Module called by that component.

2.4.2 Implementation of Semantic Relationships

A software system is a complex electronic document. It is complex because there are dependencies between components that must be maintained in order for it to be a consistent and complete document. If those dependencies are not maintained properly, the system will not function properly when it is implemented. The relationships Calls, Inputs, Outputs, Source were defined to capture the semantics of control flow and data flow. Defining the relationships was not enough, though. Constraints which captured some of the meaning intended by the relationship were integrated into the implementation of SoDA.

SoDA was implemented in the KEE system. The document model and its components were implemented as object classes (KEE Units). Behavior was implemented as methods on those units, while information and was implemented as attributes (KEE slots) defined on those units. To implement parts, an attribute called Submodules was defined on unit Module that would contain a list of the modules which were parts. An attribute called Supermodule was also defined to show the owner of a module. Associations were defined by creating an attribute for each owned relationship and an attribute for each attached relationship. So Inputs were captured by an Inputs attribute on module which contained a list of data and a Consumers attribute on data which contained a list of module. Outputs were captured by a Outputs attribute on Module and a Producers attribute on Data. Calls were captured by a Calls attribute and a CalledBy attribute on Module. The Source relationship was implemented as a method on Data.

Unit Module

Slots

Name:	symbol
SubModules:	List of Module
SuperModule:	Module
Input:	List of Data
Output	List of Data
Calls:	List of Module
CalledBy:	List of Module
Activation Rule:	Lisp List

Unit Data

Slots

Name:	Symbol
Producers:	List of Module
Consumers:	List of Module
DataType:	Symbol

Having implemented our Document Model in this manner, more care had to be taken with the actions add/remove part and add/remove relationship. Whenever a part was added or removed, the action had to ensure that both the SuperModule of the part and the SubModules of the owner were updated. Similarly, both the Inputs of the owner and the Producers of the attachment had to be updated when adding or removing an Inputs association.

To fully support the meaning intended by these relationships another step must be taken. Otherwise, actions involving relationships will result in an invalid system that will not run properly. If a component is designed to take data as input, but there is no other component providing that input, the system will not work. Constraints were created in SoDA to capture desired properties of the system like the one just mentioned. Each constraint was an action that tested the parts, information, and associations of the system, using the primitive actions already defined for the system. These were added as compound actions to the behavior for Data and Module. In addition, there was a valid? action added to each component that tested all constraints, and a valid? action for the system document model that tested the validity of all parts.

Data

Constraints:	path-sink?, primary-source?, continuous?

Module

Constraints:	activation-called?, calls-activated?, consumed-inputs?, output-source?

The constraints for Data used the idea of a *primitive* Module, which was a module that had no calls to other modules in the system. All data in the system would originate with primitive modules (which read from files or received input from users) and terminate with primitive modules (which wrote to files or sent output to users). The constraint path-sink? ensured that Data terminated, while primary-source? ensured that there was a single originating source for Data. The constraint continuous? checked that there was a continuous chain of modules passing the data from one to the other for all paths from the source to its sinks.

The constraints for Module focused on internal consistency amongst its parts. The constraint consumed-inputs? ensured that any data input to the Module was used by one of its parts (unless it was a primitive module), while output-source ensured that any data output from the Module was created by one of its parts (unless it was a primitive module. The constraints activation-called and calls-activated ensured that there was consistency between the description of the Model captured in its Activation rule, and the Calls relationships which had been defined for it with other modules.

These constraints proved very useful in focusing design work on the resolution of inconsistencies and incompleteness in a system. In chapters 3 and 4 we will describe the interaction which resulted from a design document defined this way and techniques which were used to automate it further.

2.4.3 Document Models for Database Systems

The Axiant Prototype automated the development of database systems. A database system is comprised of applications which process data in databases. A database has domains which define the types of data (e.g. data, money, address) in the database, and tables which group data into records (e.g. employee, customer, order). An application has forms for viewing and updating records, reports for listing and summarizing tables of data, and utilities for maintaining and processing databases.

The Axiant Prototype supported development of database systems as a collection of programming objects which could be created and modified through a variety of graphical interfaces. Programming objects were grouped into documents corresponding to domains, tables, forms, reports and utilities. A team of developers was able to work on a database system at the same time, sharing network access to documents.

A special notation and diagrammer were used in the project, to define the document models used for domains, tables, forms, reports, and utilities. A code generator (described in section 2.4.4) generated the code which implemented the models. Figure 11 shows a model diagram for the form document in terms of its components and owned relationships (data items and actions were specified in a pop up dialog box for each component).

A Form has Fields, Queries, and Buttons as parts. A Query accesses and updates items from tables in a database. A Field displays and modifies one of those items, while a Button is used to trigger an action like find or save associated with a Query. A Field also has Procedures and a List Lookup as parts. Procedures are used to customize the behavior associated with editing a Field, while a List Lookup is used to provide a list of possible values to choose from for a Field. A List Lookup has a Subtype part which can be one (and only one) of Check List, Radio List, or Select List. A Select List provides a drop down list of values to choose from when a Field is clicked on. A Check List or Radio List is used when there are a small number of values to choose from. Instead of a dynamic drop down list, all values are displayed statically with the field on the form as a list . With a Radio List only one value can be selected, while with a Check List any number of values can be checked.

There is an ActsOn relationship from Button to the Query which it triggers an action on. Similarly, there is an Edits relationship from Field to the Query for which it displays and modifies an Item. There can also be Sets relationships from Field to other Fields on the Form. This indicates that an action associated with the Field can cause another Field to change. There is a Filters relationship from List Lookup to the Query which it uses to create the list of possible values for a Field. There are also relationships to other documents. A Calls relationship indicate that there can be links from one Form document to other Form documents that enable one to traverse from Form to Form. A Query has an Accesses relationship to each of the Items it accesses in a database. An Item is part of a Table document.

Parts and relationships were specified and aggregated for components and document models as list attributes. Queries, Buttons, Fields, Procedures, Lookup, ActsOn, Edits, Sets, Filters, Calls, and Accesses are all lists of Components of a certain type. (e.g. ActsOn is a List of Query). Parts lists were indicated by the keyword part (e.g. Buttons is a Part List of Button). External Relationships were indicated by the keyword external (e.g. Accesses is a External List of Item in Table). It was also useful to be able to sort and index lists in order for example to see the list of Forms called by a Form in alphabetical order according to name (e.g. Calls is an External List of Form Ordered Ascending by Name). Every object also had a “parent” attribute to indicate which object or document it was part of. Every relationship actually had two attributes specified for it: one to capture the “to” link the other to capture the “from” link (e.g. in addition to a Calls attribute there would be a CalledBy attribute).

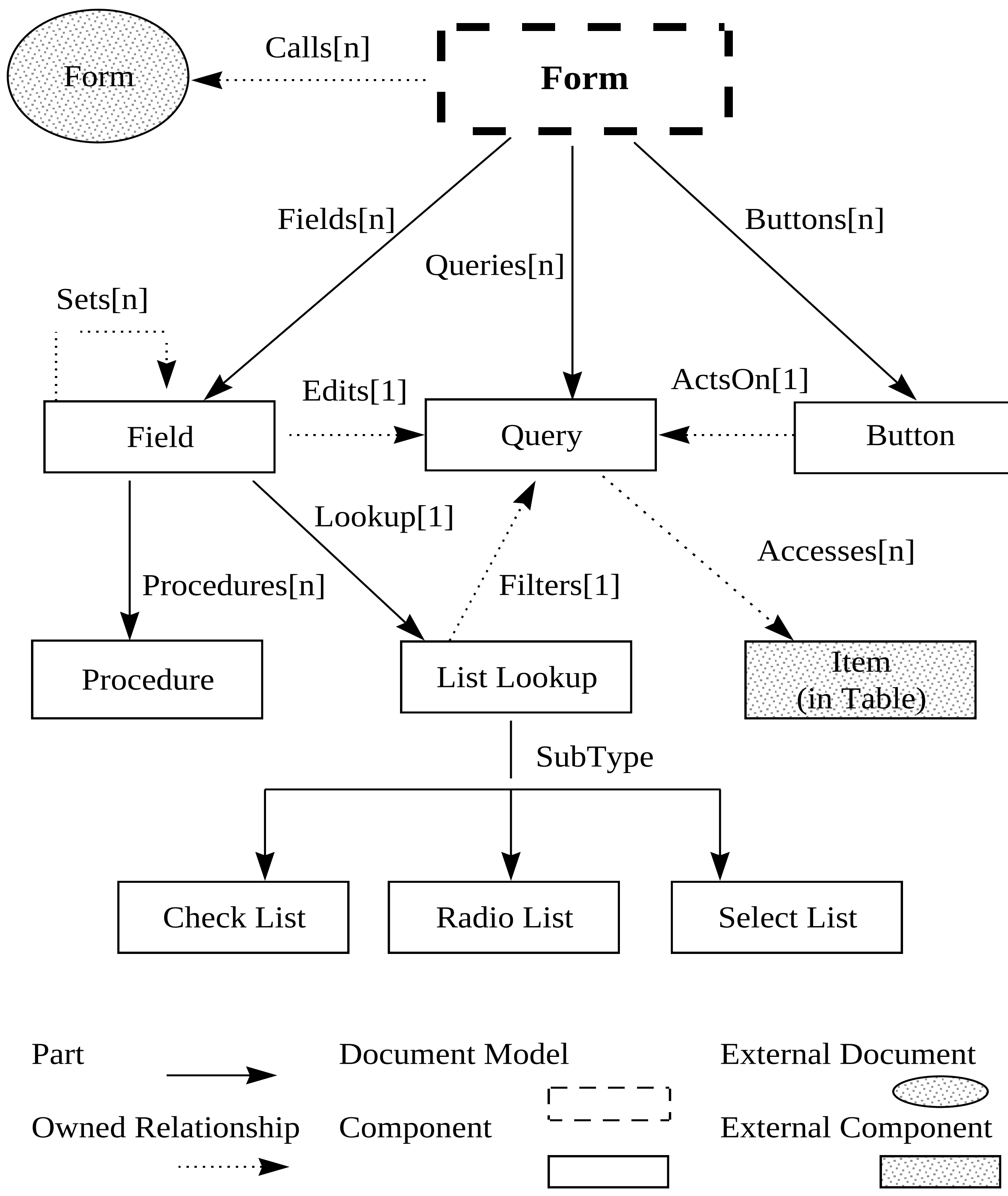


Figure 11. Form Document Model

Subtype was a special case that was handled differently to support choice in the aggregations for parts and relationships. It was implemented as an optional part, but its semantics correspond to a simple form of single inheritance. A component could have any number of components as subtypes which were optional parts. An object which was an instance of such a component would have at most one subtype part. For example, a List Lookup instance would have only one (or possibly none) of a Check List instance, a Radio List instance, or a Select List instance as a subtype. The code generator (discussed in the section 2.4.4) created special actions for components linked by subtype that supported the semantics of single inheritance.

Data items were specified for components and document models as labeled attributes using a standard set of simple types like fixed length strings, integers, large integers, floats, double precision floats, and blobs. The intention was to keep data items simple to facilitate the storage of documents in a relational database. A simple syntax was also used to specify initial values for data items, and to specify restrictions on the range of values allowed. Here is a sample of the document model for Form and the Lookup List component.

Form

Name:	String[32]	(Initial Value = “ “)
Title:	String[255]	(Initial Value = “ ”)
Queries:	Part List of Query Ordered Ascending by Name	
Buttons:	Part List of Button Ordered Ascending by Label	
Fields:	Part List of Field Ordered Ascending by Label	
Calls:	External List of Form Ordered Ascending by Title	
Calledby:	External List of Form Ordered Ascending by Title	

Lookup List

Filters:	List of Query	(Size <=1)
Subtype:	One of Check List, Radio List, Select List	

It was not necessary to specify actions for document models. A generic list of actions was automatically provided for components and document models. The code generator automatically created the code which implemented these actions. Get and set was provided for attributes corresponding to data items. Get, Delete, Append and Insert was provided for list attributes. These maintained the order specified for the list. Create was provided to create a document or object instance for document models and components. An object instance was always created as a part of a document or another object which became its owner. A semantic delete was provided to delete a document or object and its hierarchy of parts. A semantic copy was provided to copy a document or object and its hierarchy of parts. If an object was copied, it was always copied to be a part of a document or another object which became its owner. In order to facilitate persistent storage of documents, there was also a store and load action which could encode and decode an object or document into and out of a string representation.

Generic Actions

Get	Get the value of an attribute
Set	Set the value of an attribute
GetList	Get a list of objects or documents for an attribute
AppendList	Add an object or document to the end of a list for an attribute
InsertList	Insert an object or document into a list for an attribute
DeleteList	Remove all objects or documents from a list for an attribute
Create	Create an object and insert it into a part list
Delete	Perform a recursive delete of a document or objects and all its parts
Copy	Perform a recursive copy of a document or object and all its parts
Store	Encode an object or document into a string representation
Load	Recreate an object or document from its string representation

2.4.4 Code Generation

In the Axiant Prototype project, a code generator was built to automatically implement any document models created using the diagrammer and notation described in section 2.4.3. Document models for Domains, Tables, Forms, Reports, and Utilities were implemented in this manner. This proved extremely useful since each document model went through a number of iterations before it was finalized. It was also possible to add extra functionality to each document model as it became available simply by rerunning document models through the code generator. For example, support for a copy action was identified as a useful feature half way through the project. There were a number of complexities which needed to be handled by code generation. Special mechanisms were needed to handle component subtyping, external relationships, and a semantic copy and delete.

Each document model and component was represented by an object class. Parts, relationships, and data items were all attributes or list-valued attributes of those object classes. All actions were methods on the object classes. The code generated for each action supported the semantics associated with parts, relationships, data items and subtypes. For example, if an object was appended to an attribute list which represented the "to" link for a relationship, then the "from" link would be updated appropriately as well. The actions that modified links defined by relationships also maintained the order and indexing of lists automatically. Support for initial values and restrictions on values for a data item was provided by generating the appropriate code into the methods which implemented get and set actions for that data item.

Component Subtyping Versus Inheritance

Subtype was a special attribute used to support choice in aggregation. Our example of subtyping for List Lookup will illustrate how subtyping was implemented. Initially if an object instance of List Lookup was created as a part of a Field Object. An action GetType would return "List Lookup". However, an action SetType could also be invoked to set the type of the object instance to any of "Check List", "Radio List", or "Select List". If the type was set to Radio List, the object instance would support all the actions, parts, relationships, and data items defined by both List Lookup and Radio List GetType would return "Radio List", but another action GetAllTypes would return a list ("List Lookup", "Radio List"). SetType could now be invoked to set the type of the object instance to any of "List Lookup", "Check List" or "Select List". If the type of the object was changed to say Check List, then all actions, parts, links and values defined by List Lookup would remain, but those defined by Radio List would be deleted.

This implementation of Subtype should be contrasted with the usual implementation of inheritance. Switching the type of an instance dynamically in the manner we have described is consistent with the semantics of inheritance but is not usually supported in standard implementations of inheritance. It is, however, quite useful. A database system developer who decides to change a Radio List to a Check List will lose all the work that had been put into Radio List, if the system does not support a simple change of type.

Our use of Subtype also supports Check List being a subtype of another component, say Multiple Selection List. Our semantics, however, are not at all the same as the semantics usually implemented for multiple inheritance. Check List is a component which is completely independent of List Lookup or Multiple Selection List. It can be used in any document model. In this particular case, it is a subtype *part* of List Lookup. As such, it is part of the definition of List Lookup and contributes to the overall definition of an instance of List Lookup. If Check List was used somewhere else as a subtype *part* of Multiple Selection List, it would contribute to the overall definition of an instance of Multiple Selection List. However, if it was used somewhere else as a simple part without any subtyping, then an instance of Check List in that document would be defined solely by Check List without any interaction with List Lookup or Multiple Selection List. We use Subtype as a flexible mechanism for choice in defining the composition of a document model or components. During model definition, components are independent constructs that can be reused and combined.

External Relationships

In the Axiant Prototype, documents were stored in a network accessible location and shared amongst developers. Documents were converted to a string representation when they were stored and converted from a string representation into an object representation when they were loaded into memory. An entire document would be stored and retrieved at a time, so links within a document were implemented using a pointer representation. Links between documents could not be handled in this manner, because the documents or objects linked to might not be in memory or might be in use by another developer. A special module was built to handle the coordination of documents between developers and to manage and track "hypertext" style links with other documents. This module was part of the document agent framework which is discussed in chapter 3. The module created for the Axiant Prototype is describe in section 3.4.5.

The code generator provided actions which enabled the encoding and decoding of documents to and from a string representation. The actions for handling list attributes were implemented to call predefined functions in the special module in order to support external links.

Semantic Copy and Delete

The document models defined for the Axiant Prototype were intended to support a graphical interface where operations like "Cut, Copy and Paste" would be required. Copying and deleting a complete parts hierarchy is a common and tedious task, so actions for copy and delete were generated for each component and document model. Special care has to be taken in managing the relationship links, internal and external, when performing such operations. The following algorithms, we believe are fundamental to the conceptualization used in building a document model. In section 4, we will revisit this idea in the context of the SoDA project. In that project, a different algorithm for semantic copy was used which seemed more intuitive and more powerful but which proved to be less effective as an enabler for task automation.

Semantic Copy of a Document or Object

1. Traverse parts hierarchy of the object or document, marking each object in the parts hierarchy.
2. Copy the document or object and each marked object.
3. Create attributes for data items by duplicating the original attributes,
4. Create list attributes analogous to the original list attributes but use the newly created objects.
5. The list attributes for the objects in the new copy will differ from the original in that only copies of marked objects will be used.
6. However, list attributes for "To" links of external relationships should be duplicated exactly.
7. List attributes for "From" links should be ignored, they will initially be empty in the new copy.

E.g. If one was copying a Form, one would copy all objects and links within the form. One would also want to duplicate the Calls list of Forms that it links to. However, it would not be appropriate to duplicate the Calledby list of Forms which linked to the original form. Those links represent attachments made by those Forms to the original. They are not owned by the original form, and they were not intended to be linked to the newly created copy.

Semantic Delete of a Document or Object

1. Traverse parts hierarchy of the object or document, marking each object in the parts hierarchy.
2. Remove the document or object and each marked object from any list attributes of documents or objects which have not been marked. This includes all internal and external links.
3. Delete the document or object all marked objects and their attributes.

Note: That #2 requires that documents which have attached an external link to the deleted objects or documents must be updated. If a table is deleted from the database, all forms which have Querys that accessed items of that table must be updated. A conservative approach would not allow a delete to occur in such a situation.

2.5 Relationship To Other Work

2.5.1 Research

Our concept of system which we apply to documents used in knowledge work is based on work from the Delta project [Haanlykken & Nygaard 81]. The importance of parts, especially as contrasted with the use of inheritance is presented in [Osterbye 90]. A further conceptual understanding of complex assemblies is presented in [Kristensen 94]. Our approach to document modeling has been strongly influenced by entity-relationship modeling [Chen 77]. Some of the issues which arise in differences between data modeling approaches to subtyping and the classical use of inheritance in object-oriented approaches are discussed in [Thomann 94]. A detailed discussion of the distinction between a component-based approach and a class-based approach is presented in [Wozniewicz 94]. In [Jacobsen et al 90] a distinction is made between entity, interface, and control objects. In many ways, our document models focuses on organizing entity concepts underlying a domain of knowledge work.

Similar approaches to modeling documents have been taken using grammar-based approaches in the Cornell Program Synthesizer [Reps 84], Muir [Winograd 87], and the SGML standard [Goldfarb & Rubinsky 90]. The Refine programming language which originated with work on the CHI system [Phillips 83] combined a grammar-based approach to modeling with knowledge-based techniques to automate the implementation of formal specifications. While the notation of grammar-based approaches is quite different from ours, the essential difference is the lack of support for modeling actions and semantic relationships between components. Hypertext system capture semantic relationships between documents using concepts of nodes and links. Work on the HyperPro system [Osterbye & Normark 93] to support software development using typed nodes and links has strongly influenced us. [Normark 89] discusses general issues in providing "hooks" and "open points" that provide a background on our discussion of enabling extensions and customizations of actions when document models are implemented. The Beta language [Madsen et al 93] defines an "inner" procedure concept which is convenient for defining pre and post actions.

2.5.2 Industrial Practice

Notations and diagramming tools for creating models as a basis for implementing software have become quite common place in the industry. Tools like IEW from Knowledgeware based on data modeling and Rose from Rational based on object modeling are representative and have influenced our work.

Industries are now attempting to establish standards for notations and define specific models in order to enable automation. SGML defines an ISO standard (ISO 8879-1986) for document modeling. It primarily define a common storage format which can be used to exchange information in documents independent of the applications which use it across most operating systems. It deals primarily with ASCII text and there is no standardization for defining actions or relationships. HTML is closely related to SGML and it defines a single document model which has been used as a basis for development of the World Wide Web. The CALS initiative [Walter 92] is one of the largest attempts to automate based on the SGML standard which focuses on technical manuals in the United States Department of Defense.

CMISE and SNMP define standard protocol for communicating with software that manages information [Stallings 93]. Those protocols are based on defining an object model for the managed information in a standard notation (e.g. ASN.1). There are similarities with our approach to document models. The essential step in CMISE is the definition of a containment hierarchy of object classes which is the equivalent of our parts hierarchy of components. Data items, and relationships are specified as attributes for which Get and Set methods are defined. Other Actions are defined by methods. The telecommunications industry has defined specific models (e.g. CCITT - M.3100 for network elements) as a basis for automating management of telecommunication networks.

3.0 Document Agents

In this thesis, we approach software automation as the creation of an electronic domain in which the exchange of information fundamental to knowledge work can take place. In our architecture, this software is created as a document agent which embodies information in documents defined by document models, and performs actions that exchange information on behalf of participants in a domain of knowledge work. In this chapter, we define our document agent architecture specifying the functionality which is required for automation and discussing design issues relevant to its integration into a domain of knowledge work

Our aim is to use a document agent as a basis for improving the level of interaction provided to participants by software automation. To do so, a document agent must provide an environment which mirrors the conceptualization of knowledge work that is understood by participants. Document models should structure data into documents as a reflection of the manner in which participants conceptualize information relevant to work. The actions performed by a document agent should be organized as a reflection of the manner in which participants organize their work into tasks. At the same time, an electronic domain offers new potential for managing information and enabling its exchange. As a result, one should expect that the way in which we conceptualize and organize our work will evolve as a process of automation.

3.1 Electronic Domains

"' ... I'll tell you all my ideas about Looking-glass House. First, there's the room you can see through the glass - that's just the same as our drawing-room, only the things go the other way ... But oh, Kitty! now we come to the passage. You can just see a little peep of the passage in Looking-glass House, if you leave the door of our drawing-room wide open: and it's very like our passage as far as you can see, only you know it may be quite different on beyond ...'"

Alice in Through the Looking Glass, from The Best of Lewis Carroll (Secaucus, N.J. 07094: Book Sales Inc.), p. 173-175

3.1.1 The Vision

It is a premise of this thesis that documents convey knowledge, because they organize data in a form which humans can interpret in a useful manner. Knowledge work is centered around documents as a system of information exchange. A particular domain of knowledge work can be characterized by its documents and the tasks performed with those documents. An electronic domain, which we define as *the embodiment of a domain of knowledge work in software*, enables the automation of work because it provides *a domain of information exchange* in which knowledge is conveyed.

A domain of information exchange is characterized by the types of documents supported and by the tasks which can be performed as information exchanges. Document models define each document type as a system of information exchange. The execution of those models by a software process, which we call *a document agent*, creates documents that participate in information exchanges. The state of these documents, at a particular moment in time, represents the state of knowledge work in the domain. Tasks are transformations of that state which are enacted by a document agent as information exchanges involving documents.

3.1.2 Scenario

Let us consider a scenario which illustrates our understanding of document agents. Figure 11 is an illustration of a particular situation half way through the scenario.

A banker receives a late payment on a loan from a client, Fred Smith, along with a change of address notification. The banker pulls up a Client Form on his computer, types the last name Smith and presses Find. There are three Smiths in the system (Fred, Jane, and Tom) and the banker is prompted to select one. The banker selects Fred. The Client Form displays Fred's information and the banker proceeds to change the address. From the Client Form, the banker also pulls up a Payment Form to record the payment on the loan. The banker presses OK and the payment is entered into the system.

Returning to the Client Form, the banker notices that the client has been identified as a credit risk due to the late payment. Realizing that the payment was late due to the clients move, the banker attempts to change the credit risk entry. A message is displayed to inform the banker that he does not have authority to change the entry, but that he can make a request to the credit manager to perform the change using a Request Form. The banker proceeds to fill in the Request Form asking that the credit risk entry be changed.

At that moment, the power goes out on the banker's machine. He accidentally kicked the plug out with his foot. As he powers his machine up, the credit manager walks by. The banker explains the credit risk entry to the credit manager, and she promises to look after it. She returns to her computer and pulls up the Client Form and Payment Form for Fred in the same manner as the banker. She changes the credit risk entry.

Figure 11 illustrates the situation just after the bankers machine has powered up. During power up, the system displayed a message which asked the banker if he would like to resume his interrupted work session? When the banker says yes, he was returned to the Request Form. Since the banker has already spoken to the credit manager, he selects Cancel on the Request Form. Meanwhile the credit manager, presses OK on the Client Form, completing the change to Fred Smith's credit risk entry. Since the task associated with Fred Smith is finished, both the credit manager and the banker move on to other business.

If we view the software which is used by the banker and the credit manager in their work in terms of a document agent architecture, some observations can be made. The *execution* of a document agent creates documents. As soon as the plug is pulled, there is no document for the banker to interact with. The *state* of documents created by a document agent reflects the state of work at the bank for both the credit manager and the banker. In order to determine the state of the client's situation at the bank, the banker and the credit manager search through existing documents and find the information which characterizes that state in documents relating to the client. Finally, a document agent is used to coordinate the *transformation* of state captured in documents. The changes made in a document are not complete until OK is selected. The changes can also be discarded to return a document to its original state by selecting Cancel.

The document agents supporting the knowledge work at the bank provide a domain of information exchange in which the bank's knowledge work can occur. The execution of a document agent provides state and transformations of state that conveys knowledge used by the banker and the credit manager in their work.

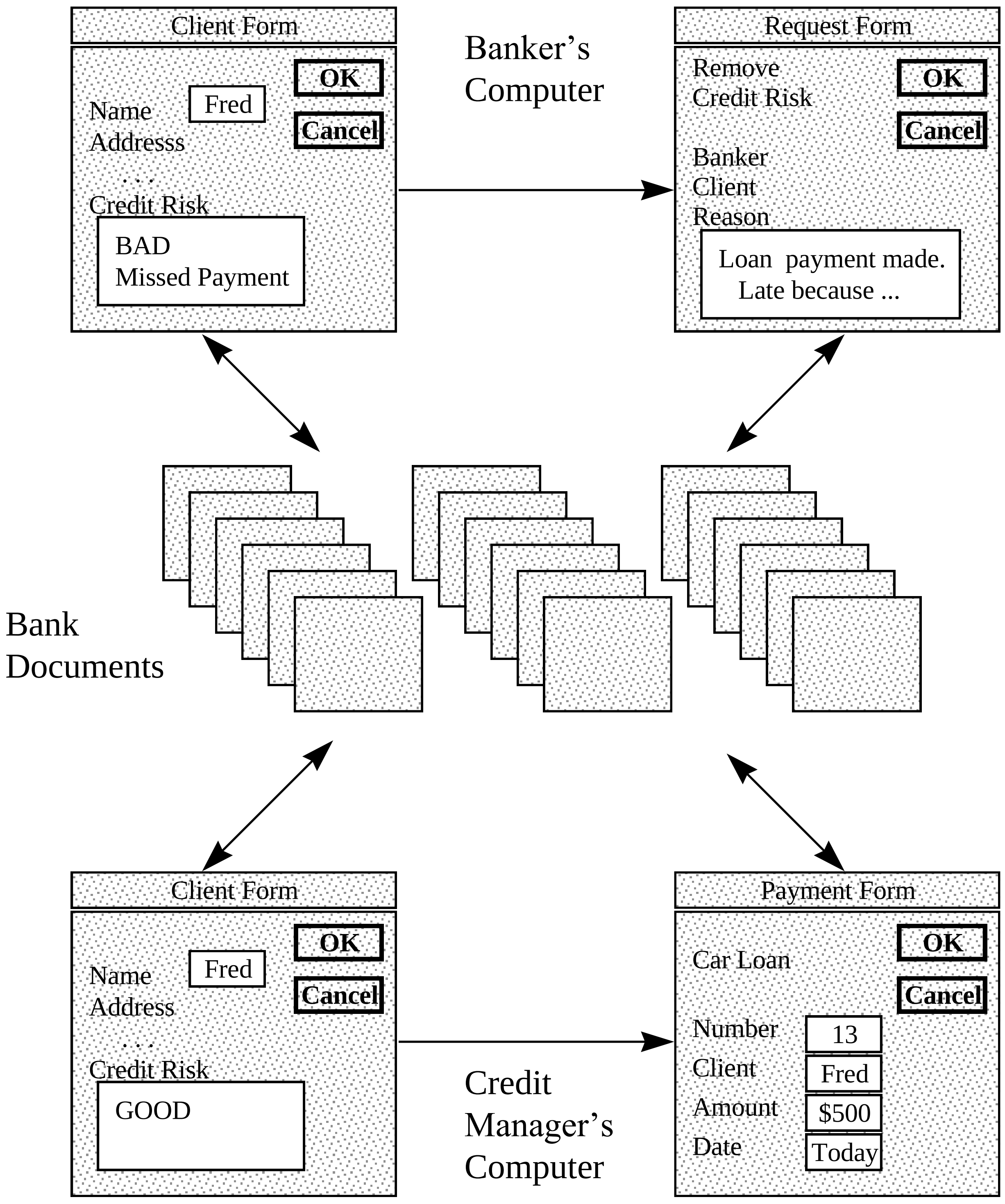


Figure 12. Scenario as Banker's Computer Resumes

3.2 A Domain of Information Exchange

In this section, we will elaborate the internal architecture of the document agent specifying the functionality required for automation. Our intention is to mirror the manner in which participants conceptualize and organize their work within a particular domain. In doing so, we take an approach which is similar to approaches taken to the architecture of object-oriented systems [Jackson 83] [Mathiesen et al 95]. We separate out the interfaces to the document agent, the organization of behavior into tasks, and the organization of information into document models.

The premise behind our approach is that the level of interaction provided by a document agent is a function of the effort required by participants to interpret the data and data processing of the computer in terms of their work. Figure 13 illustrates the internal architecture of our document agent showing how its organization is intended to reflect the work performed by participants.

Participants in a domain of knowledge work define documents to organize information and they define tasks to organize their work around actions performed with documents. Document models within the document agent define the organization of information for different types of documents (and the information exchanges they support) independent of any particular task. Automated tasks within the document agent define the actions which a participant can request as they perform their work. These actions interact with different document models to trigger information exchanges in documents. Interfaces within the document agent define the mechanisms by which participants can interact with a document agent. Usually this includes support for a graphical display.

Interfaces also define the mechanisms by which a document agent can interact with other software processes (including other document agents). Usually this includes support for a persistent storage facility like a file system or a database, from which documents are retrieved and filed. The nature of electronic media entails special requirements on the document agent for state management and transformation control in order to support and coordinate knowledge work. A special task for state management controls the interface to persistent storage in a manner that allows participants to organize the state of knowledge work into collections of documents. A special task for transformation control manages the triggering of information exchanges in documents by other tasks.

Fundamental to the document agent architecture is the ability to add, replace or customize interfaces, tasks, and document models. Different operating systems and implementation languages will support different mechanisms for supporting this feature. Rather than adopting a particular implementation mechanism we have restricted ourselves to specifying the functionality required and identifying design issues relevant to implementation. We discuss some of the design issues in section 3.3.

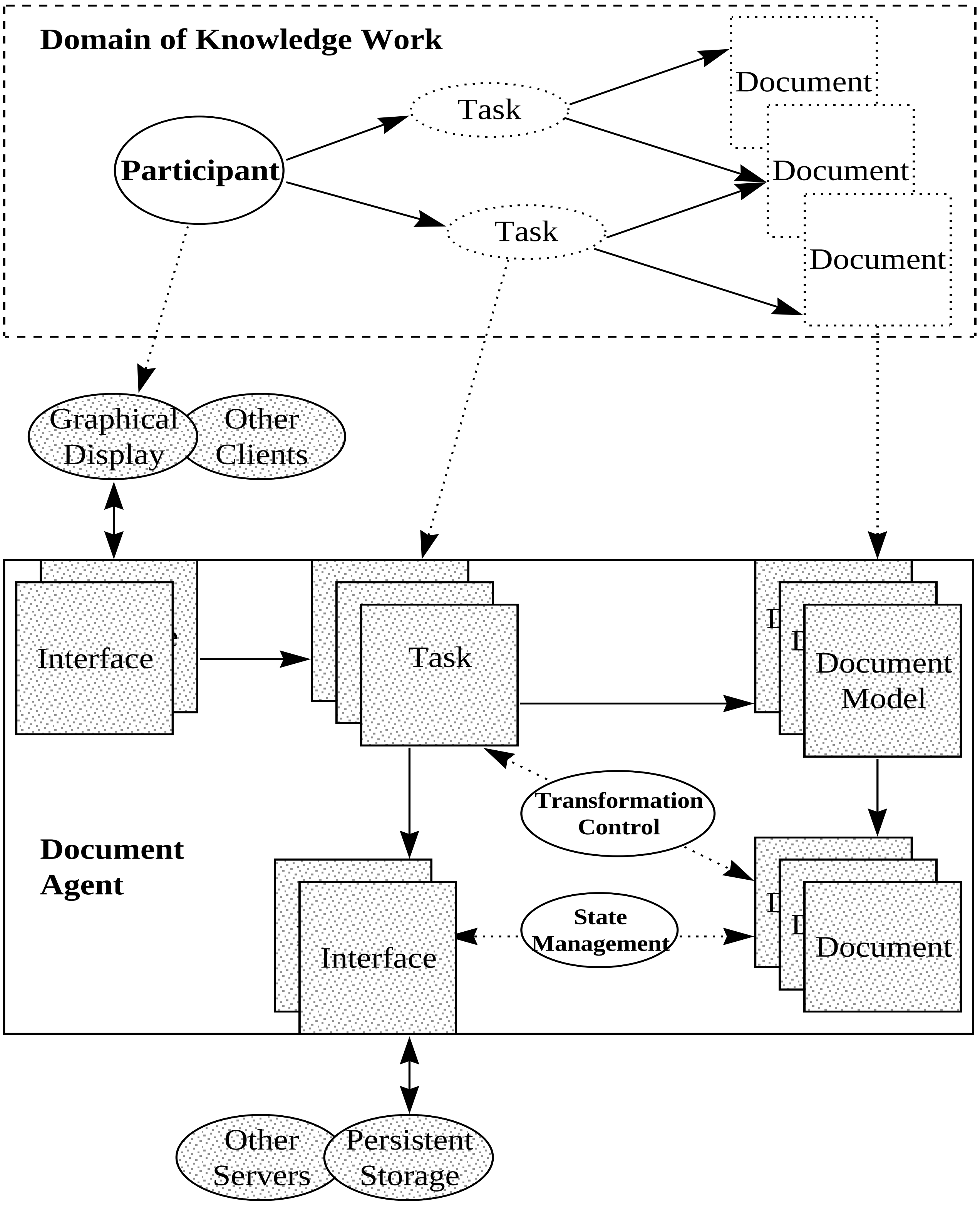


Figure 13. Understanding Document Agents

3.2.1 Execution Environment

A document agent is a software process whose execution on behalf of participants creates a domain of information exchange.

A domain of information exchange consists of *document models* which define the types of documents that can be created, *tasks* which define the information exchanges which can occur, and *interfaces* which enable information exchanges to be requested. The execution of a document agent is tied to the particular hardware and software environment(s) in which it operates. Within a document agent, there is an abstraction layer in which document models and tasks can be implemented independently of those environments. That abstraction layer provides an environment for creating a domain of information exchange.

Document Models

A document agent executes document models relevant to a particular domain of knowledge work. The triggering of information exchange events by the document agent creates documents consisting of objects, values and links. The implementation of a document model is tied to the particular mechanisms supported by the document agent for triggering events. The document agent specifies the mechanisms for calling actions and identifying data items and components that are inputs and outputs. The document agent also restricts the types of values allowed for data items. Any document model implementation which conforms to this framework can be executed by the document agent. A document agent may execute many document models, one for each type of document. A document model may be executed by many document agents, although the tasks performed by each agent may be different.

Tasks

The document agent executes information exchanges associated with tasks relevant to a particular domain of knowledge work. These information exchanges are defined by actions that trigger events in one or more document models. The discussion of action for document models in section 2.2.4 also applies to actions associated with tasks. Tasks and their actions are defined separately from document models. The mechanisms used to create tasks and their actions may or may not be different from those used to create document models. Information exchange requests received by the document agent cause it to perform actions from the associated task. Special actions may create information exchanges by making requests to other software processes. The implementation of a task is tied to the particular mechanisms supported by the document agent for triggering events in document models as well as for making and receiving requests. Any task implementation which conforms to this framework can be executed by the document agent.

Interfaces

The document agent executes interfaces through which it can exchange information with client or server processes. These processes may support hardware devices such as graphical displays through which a participant can interact with a document agent. They may also support media such as a file system or a database where documents can be stored persistently. Interfaces enable information exchange in the form of requests and responses which are received and sent. The implementation of an interface is tied to the mechanisms which exist in the particular hardware/software environment where the execution of the document agent is taking place. Any process which conforms to those mechanisms can interact with the document agent. Requests are made to the document agent to perform some action associated with a task. The document agent may send zero, one, or many responses back depending on the response and the results of any action that was performed. A client process can also request to receive response(s) whenever certain actions are performed by the document agent. An action associated with a task can also make requests to server processes. Those requests will be delivered by the appropriate interface and any responses will be returned back to the task action which made the request.

Example

To illustrate the execution of a document agent, let us consider the change to the credit risk entry for Fred Smith in our scenario. The banker is sitting in front of a graphical display with keyboard and mouse. The click of the mouse and the touch of the keys on the keyboard sends a request to the document agent that the banker would like to change the entry in the Credit Risk Field. An interface in the document agent receives the request from the graphical display. The action of changing the entry is associated with an automated task in the document agent responsible for entering client information. The request could be handled by triggering the SetEntry Event for the Credit Risk Field Object defined in the Form Document Model. This would change the entry and return the new value which could be sent back to the graphical display as a response. Instead the Display Event for the Change Request Form is triggered, after it is determined that the banker is not allowed to change the entry himself. (The manner in which this is determined is discussed in section 3.2.2). The information associated with the Change Request Form is sent back to the graphical display as a response.

3.2.2 State Management

> A document agent is a software process whose state, captured in documents on behalf of participants, defines the state of a domain of information exchange.

The state of a domain of information exchange consists of the documents ant their objects, values, and links which exist in that domain at a particular moment in time. That state has *persistence*, in that it will not change over time unless an information exchange is enacted by the document agent. The state of the domain of information exchange reflects the state of the domain of knowledge work, and as such the *organization* of the state into documents is accessed and updated by the document agent on behalf of participants. Knowledge work may involve more than one participant, each with *accountability* for different tasks. The document agent will manage the access to the state of the domain through the definition of accounts which link access to a participant. State management can be centralized in a single task which controls the interaction with a server process or processes that provides a persistent storage facility.

Persistence

The state management task provides actions to save documents, which exist in the execution of the document agent, into some facility for persistent storage and to load documents, into the execution of the document agent, from that facility. There are a variety of approaches that can be taken: a binary dump to file, a translation of the document into databases or fielded ASCII files.[5] The state management task handles all requests and responses through the interface to that facility, including any conversions to and from the storage format. The framework implemented by the document agent may require actions to be provided in document models to support this conversion.

If documents are large, then it may not be practical to load the entire document into the execution of the document agent. The state management task is responsible for loading and saving parts of documents as necessary. A variation of the approach used to define deep copy and delete in section 2.4.4 is a starting point for implementing such a mechanism.

Organization

The state management task provides actions to organize, and access documents which are active in the execution of the document agent or which have been saved into a facility for persistent storage. These actions provide an interface through which the organization and access required for knowledge work is mapped to the mechanisms provided by the facility for persistent storage. In some cases this mapping may be direct (e.g. each document is a file in the file system and the mechanisms for organization are exactly what the file system provides). In other cases, the state management task may provide extra functionality (e.g. documents have descriptions attached to them, and documents can be retrieved by description). The organization supported by the document agent can be defined and implemented in a document model (with components like directories, and actions like retrieve by description) with which the state management task interacts.

The actions for document access also support external links between documents. The document agent framework will provide mechanisms for referencing other documents that document models will use in defining external relationships and actions which access them. A task which accesses a document through an external link will request that the state management task locate the document, and if necessary, load it from persistent storage. (In some cases, the document may be controlled by another document agent, in which case the task will need to interact with the document by means of requests to that document agent).

[5] The approach taken depends on the type of information being stored and the use it is put to. Binary dumps are simple and fast, relational databases allow for sophisticated querying, sharing and robust backup, while ascii files allow for version control and readability of information that is largely text based.

Accountability

The state management task provides actions to assign accountability for individuals or other software processes which make requests to the document agent. These actions are used to create accounts that are registered with the document agent. These accounts are used to track requests and protect the security of the state of knowledge work. Actions are only performed on behalf of accounts which have authority to request those actions. The accountability supported by the document agent can be defined and implemented in a document model (with components like accounts, and actions like check authority for a request) with which the state management task interacts.

When several individuals are at work in the domain, accounts also serve to distinguish participants. The state management task can provide actions which identify or prevent actions initiated by two different accounts which would conflict. The actual mechanisms for managing potential conflicts will depend on the domain of knowledge work. In some domains, it is important that no two participants work with the same document at the same time. In other domains, shared access to documents is allowed as long as the transformations of state enacted by the two participants can be merged. The storage facility may have mechanisms for controlling access which can be used by the state management task. (e.g. if the facility used is a document management system with mechanisms for configuration management).

Example

To illustrate the state management task of a document agent, let us consider again the change to the credit risk entry for Fred Smith in our scenario. When the banker filled in the Payment Form and clicked on the OK button a request was sent to update the Payment Record Query for the form. Actions associated with the task responsible for entering client information trigger events which update the query with the information contained in the Payment Form. An action for state management is also invoked in order to save the information in the Payment Form to persistent storage.

When the credit manager clicks on the Payments button of the Client Information Form, a request is sent to the document agent. Actions associated with entering client information, trigger an event in the Client Information Form which returns an external link to the Payment Form. An action for state management is then invoked which locates the Payment Form and loads the information from persistent storage. A response is then returned to the graphical display of the credit manager which shows that the payment has been made as entered by the banker.

In order to manage accountability, whenever a request is received from either the credit manager or the bankers machine, an action for state management is invoked to check the authority of the request. This action triggers an event in the appropriate Account document. In the case of the banker, the check indicates that the banker does not have authority for the request to change the credit risk entry.

3.2.3 Transformation Control

A document agent is a software process whose transformation of documents on behalf of participants, defines transformations of state for a domain of information exchange.

A transformation of state for a domain of information exchange is performed by a document agent in the context of tasks which are the responsibility of participants in a domain of knowledge work. A new state of information is created from an initial state by the *composition* of information exchange events into a single compound event[6]. A document agent can support the *recovery* of the new state (if the task is interrupted before it is saved to persistent storage) as well as the initial state (if the new state is deemed not appropriate to the task). It can also support the *synchronization* of transformations which are performed simultaneously. Transformation control can be centralized in a single task which manages the triggering of events in document models by other tasks.

Composition

The transformation control task provides actions to define a transformation as a composition of events into a single compound event. Every time a task triggers a single event defined in a document model, the transformation control task records the occurrence and completion of the event as a *primitive transformation*. The event, as defined in the document model, may be a primitive or compound event. The transformation control task can also record a *composite transformation* as a sequence of transformations. In the simplest case, each request to the document agent from a client process will be recorded as a transformation. The composite transformation starts when the request is made and completes when all the resulting primitive transformations have completed It is essential that when a single request is made, either the request is not performed and the state remains the same, or all the events within the composite transformation occur as a single compound event and the new state is achieved. A transformation is a compound event that is meaningful to the participants in a task.

The transformation control task can also provide actions which modify the composition of transformations. Tasks can register pre-actions or post-actions which should be performed whenever an event occurs or is about to occur. Those actions will be invoked by the transformation control task just before or just after the event occurs in order to modify the composition of a transformation. A task could also register to block completely the triggering of the event. In this manner, the transformation control task can mediate transformations of state by several tasks.

Recovery

The transformation control task provides actions which enable either the initial state of a transformation or the new state created by it to be recovered. The actions which enable recovery can be based on versioning, replay, or undo. In versioning, whenever a transformation is started, new versions of documents are created in which events occur, so that the initial state is preserved in the previous versions. The previous versions define a checkpoint that can be returned to if the new state is abandoned. A series of checkpoints may be defined for a composite transformation. At each checkpoint, versions are saved to persistent storage. If the entire interaction with an individual or software process is defined as a composite transformation, then any of the states up to the last checkpoint written to persistent storage can be recovered should that session of interaction be interrupted prematurely.

[6] The end result of a transformation can be described as a state transition. We use the term tranformation to describe the events which result in a state transition, especially since the state is captured in documents. The term transformation is more usual when referring to changes reflected in a document.

Replay can be used for recovery if the transformation control task saves a transformation record to persistent storage whenever a primitive transformation completes. If the execution of the document agent is interrupted prematurely during a composite transformation, the document agent can replay the recovery log of transformation records, triggering each primitive transformation again, up to the end of the last completed primitive transformation.

Undo can be used for recovery if the transformation control tasks records the primitive events that occur within a primitive transformation. Each primitive event must have associated with it an “undo” action that will reverse the changes to state caused by the primitive event. One can undo a transformation, recovering its initial state, by replaying the transformation record in reverse, performing the undo action associated with each primitive event.

Synchronization

The transformation control task provides actions to define simultaneous transformations and synchronize the effects of those transformations. Composite transformations can be defined which segment the actions of the document agent into simultaneous transformations. A composite transformation can be composed of all the primitive transformations applied to a single document, or all the primitive transformations triggered by requests associated with a single account. If the document agent has more than one document or more than one account active at a time, then there will be simultaneous transformations. The events triggered by one transformation can conflict with the events triggered by another transformation (for example, if one account requests that a document be deleted while another account is modifying the document).

A set of transformations, starting from an initial state, is synchronized if one can define a single composite transformation, starting from that initial state, that interleaves the primitive transformations of each member of the set in such a manner that no *conflicts* occur. A conflict occurs if this synchronizing transformation can not be replayed completely from the initial state. In the case where a transformation associated with one account deletes a document while another transformation associated with a different account modifies the document, a synchronizing transformation could be created by deleting the document after the modifications were made. Obviously, the manner in which a synchronizing transformation should be defined is dependent upon the nature of a particular domain of knowledge work and the perspective of its participants.

The transformation control task enables simultaneous transformations to be synchronized by providing actions to prevent *potential conflicts* from occurring and to notify participants when they do occur. A potential conflict occurs, if there is any composite transformation that can be defined which interleaves the primitive transformations of simultaneous transformations in such a way that a conflict occurs. Potential conflicts can be prevented by versioning documents. Whenever a transformation is taking place that will change a document, a new version of the document is created. Only events triggered within this transformation will be able to change the new version. Events in other transformations can access the old version of the document, but are prevented from changing the document by the transformation control task.

The prevention of potential conflicts in this manner may not be practical for a particular domain of knowledge work, if, for example, transformations take place over a long period of time. A more flexible mechanism is to allow several transformations to change a document simultaneously. A copy of the current version of the document can be created for each transformation that is triggering changes to the document (along with a notification that conflicts may occur). When one of the transformations completes, a new version of the document is created by replaying the transformation with the current version of the document. If a conflict occurs, then the document agent will revert back to the initial state of the current version and the account or task which triggered the transformation will be notified.

Example

To illustrate the transformation control task of a document agent, let us consider again the change to the credit risk entry for Fred Smith in our scenario. When the banker filled in the Payment Form, there was a whole series of events that occurred before the banker clicked on the OK button (the Client name, loan, and amount of payment were each entered on the form). The document agent performing these actions treated them as a single composite transformation which completed when the banker clicked OK. If the banker had clicked Cancel, then the initial state would have been recovered in which the transformation was started. That is, the state would have no Payment Form as a record of Fred Smith's loan payment.

When the banker accidentally turns the power off on the machine, he is in the middle of a composite transformation in which he is filling out a Request Form. The document agent replays the composite transformation starting from the last checkpoint where the state was saved. So, the Client Form and the Request Form are displayed as they were when the power went off, except that the field that the banker was entering when the power went off is blank.

When the credit manager brings the Client Form and the Payment Form up on her screen, she is in the middle of a composite transformation that is occurring simultaneously with the composite transformation that the banker has resumed. The banker is viewing the Client Form and changing the Request Form while the credit manager is changing the Client Form so no conflicts are occurring.

3.3 Design Issues

We have defined a document agent as a domain of information exchange. Our intention in defining a document agent in this manner, is that the domain of information exchange will be designed to create an environment which supports and automates knowledge work. A document agent creates documents and performs tasks in its environment on behalf of the participants in a domain of knowledge work. There are a number of issues which will arise in attempting to create an environment which supports knowledge work in this manner. For the most part, these issues relate to the role of the document agent within the domain of knowledge work and the manner in which it's role can be adapted as the domain of knowledge work evolves.

Establishing the role of the document agent is primarily one of boundaries. What behavior will be located within the document agent, how will it be accessed, and what information will be captured in the form of documents? Implementing the answers to these questions will require a reorganization and evolution of the domain in which knowledge work has been taking place. The issues of determining the location of behavior and capturing information in documents will also occur in designing the parts of a document agent: its document models, tasks and interfaces.

The adaptation of the document agent as the domain of knowledge work evolves is primarily an issue of configuration. If the tasks and document models within a domain are evolving, how easy is it to change and replace the software which implements them within the document agent? This also raises the issue of portability for the document agent. Can it be relocated into a different software/hardware environment and can it's interfaces be adapted to different clients and servers using different protocols? As a domain of knowledge work evolves, it will also become an issue of how to extend and customize a document agent by adding to its existing parts.

3.3.1. Boundaries

For each participant in a domain of knowledge work, and for each part of a document agent, it is important to consider what behavior and information will be its responsibility, and to consider how it will interact with other participants and parts. When a participant interacts with a document agent through a graphical display, there is a decision to be made as to whether the graphical display will be managed by a separate process that sends requests to the document agent, or whether the graphical display will be managed by an interface within the document agent that will interpret each event in the display as a request to be transmitted to a task. If the interface is complex (as most graphical interfaces are) and there is behavior that is not central to the document agent (the layout of the display, or the display of information not contained in documents controlled by the document agent), then it is more appropriate to locate that behavior in a separate process.

There are similar considerations in determining what behavior to locate as tasks within a document agent, and when the behavior within a document agent should be divided into separate tasks. A document agent could consist of a single task that simply triggers events defined in document models. In such a situation, a lot more communication is required with client processes and a lot more behavior specific to tasks must be defined in client processes. A separate task should be created when a meaningful subset of work of reasonable complexity, identifiable by participants in the domain, is located within the document agent

There are similar issues when considering server processes like persistent storage. Actions for a task could handle their own simple file-based save and retrieve for documents they interact with. Usually, it is more appropriate to address the issues surrounding persistent storage in a single task dedicated to that functionality. More sophisticated management of state can also be provided by interacting with specialized processes for state management such as version control or configuration management systems.

3.3.2 Configuration

A domain of knowledge work will evolve over time, so it is important to understand how the behavior of a document agent can be configured or customized to adapt and evolve as well. A document agent should be able to substitute new versions of document models, tasks, interfaces or add additional ones. We can distinguish whether this is possible dynamically (without any change to the rest of the document agent) or statically (the document agent needs to be rebuilt in order to incorporate the new or changed part). Realistically, one must expect that substituting a new document model will affect tasks which use it, and that changes to tasks will affect the participants interacting with tasks. It is this interrelationship which enables the substitution of parts to adapt the document agent to different or changing environments. If the bank, decided that they would like scanned pictures incorporated into some of their form documents, then they could use a new version of the form model. Tasks and interfaces which wanted to take advantage of this new feature would need new versions as well. If this can be done dynamically, then the disruption entailed is minimized.

The ability to customize could be supported at a finer grain of detail as well. If document models can integrate new subtype components dynamically, then the bank's desire for scanned pictures on their forms, could be addressed. defining a Picture Field component as a subtype of Field. A facility for adding or modifying actions dynamically (through the use of a scripting language) is another effective mechanism for customization. Behavior is a result of actions defined in document models and tasks or the requests and responses of a participant interacting with a document agent. With a scripting language behavior performed by a participant can be automated by adding or modifying actions associated with tasks in the document agent.

Portability becomes an issue, when it is not the parts of the document agent which change but the hardware/software environment in which it operates. If the execution environment provided by a document agent for its document models, tasks and interfaces is portable across hardware/software environments, then they should be portable as well requiring only that their implementations be recompiled or regenerated. Realistically, if the mechanisms for interacting with other participants change significantly, then the document agent will be adapted by substituting or extending the appropriate interface components. Or if a participant that provides a service to a task in the document agent is substituted or enhanced, it may be necessary to substitute or enhance the task. For example, if the state management task used one type of database for persistent storage and then switched to another type of database, the task might have to be adapted.

3.4 Project Examples

In this section we illustrate our document agent architecture with example systems implemented in our project work. In the SoDA system (introduced in section 1.4.1), the architecture supported the design of software systems by focusing on the transformation of a system document which represented the design. In addition to automating design transformation, the document agent included automated tasks to ensure that constraints associated with the design were satisfied, and to suggest transformations that could be enacted to fix constraint violations when they occurred. We will show how a document model facilitated transformation control by organizing transformations and constraint violations into a flexible history list mechanism and a to do list.

In the Axiant Prototype (introduced in section 1.4.2), the architecture supported the development of data base systems by teams of developers sharing documents stored in a central repository. Mechanisms were provided to support flexible undo, recovery of work in case of system failure, and coordination of access to the documents in the central repository. We will show how a document model facilitated state management and discuss some of the implementation issues involved in ensuring a high degree of configurability and portability. This included support for third-party configuration management tools.

3.4.1 A Transformation Control Task for Interactive Design

The SoDA system created an electronic domain in which a software system could be designed. In that domain, the state of work was reflected in two electronic documents. One that defined the design of the software system and another that recorded the design session, keeping a History list of design actions taken and a To Do list of design constraint violations which still needed to be resolved.

Figure 14, diagrams the document agent architecture that was implemented to create the electronic domain. A designer interacted with the document agent through three different graphical user interfaces which were client processes. All three processes interacted with the document agent through the same client interface. A control flow diagrammer and a data flow diagrammer presented a graphical view of the software system as described in section 2.4.1. A design assistant enabled a participant to organize the design actions they requested in a History list. Constraint violations which resulted from design actions were tracked in a To Do list. The design assistant also provided suggestions on how to resolve constraint violations.

The document agent had two document models. The System model, which was described in section 2.4.1, defined a system design document in terms of Module and Data components. The Session model recorded a design session in terms of History, Activity, Transformation, Event, ToDo, and Violation components. The History component records actions performed on behalf of the designer. The designer can organize these actions on the History list by activity. An Activity component which was part of the History component enable this feature. Each user request within an activity was represented in a Transformation object, and within each Transformation. the primitive events which occurred in the system document were represented by Event objects. The ToDo component recorded all constraint violations which resulted from design actions. Each violation was represented by a Violation object. The Violation object recorded the Module or Data object where the violation occured and the particular constraint which was violated. A design was not considered complete until all constraint violations had been resolved. When a design action fixed a violation, then the corresponding Violation object would be removed from the ToDo list and deleted.

The Design Assistant in figure 14 shows a design session where support for input and output is being added to our BlackBoard system from section 2.4.1 The main activity for the session is HandleIO. As part of that activity, there is an activity to create an IO module, an activity to incorporate input and output data flow into the processing of the blackboard, and an activity to fix a constraint violation that occured in module Bbupdate. At the moment there are constraint violations associated with Data objects InputData and OutputData. The designer has not finished defining the data flow. There is also a constraint violation in module Bbupdate. Its activation rule needs to be updated to reflect the new data and control flow.

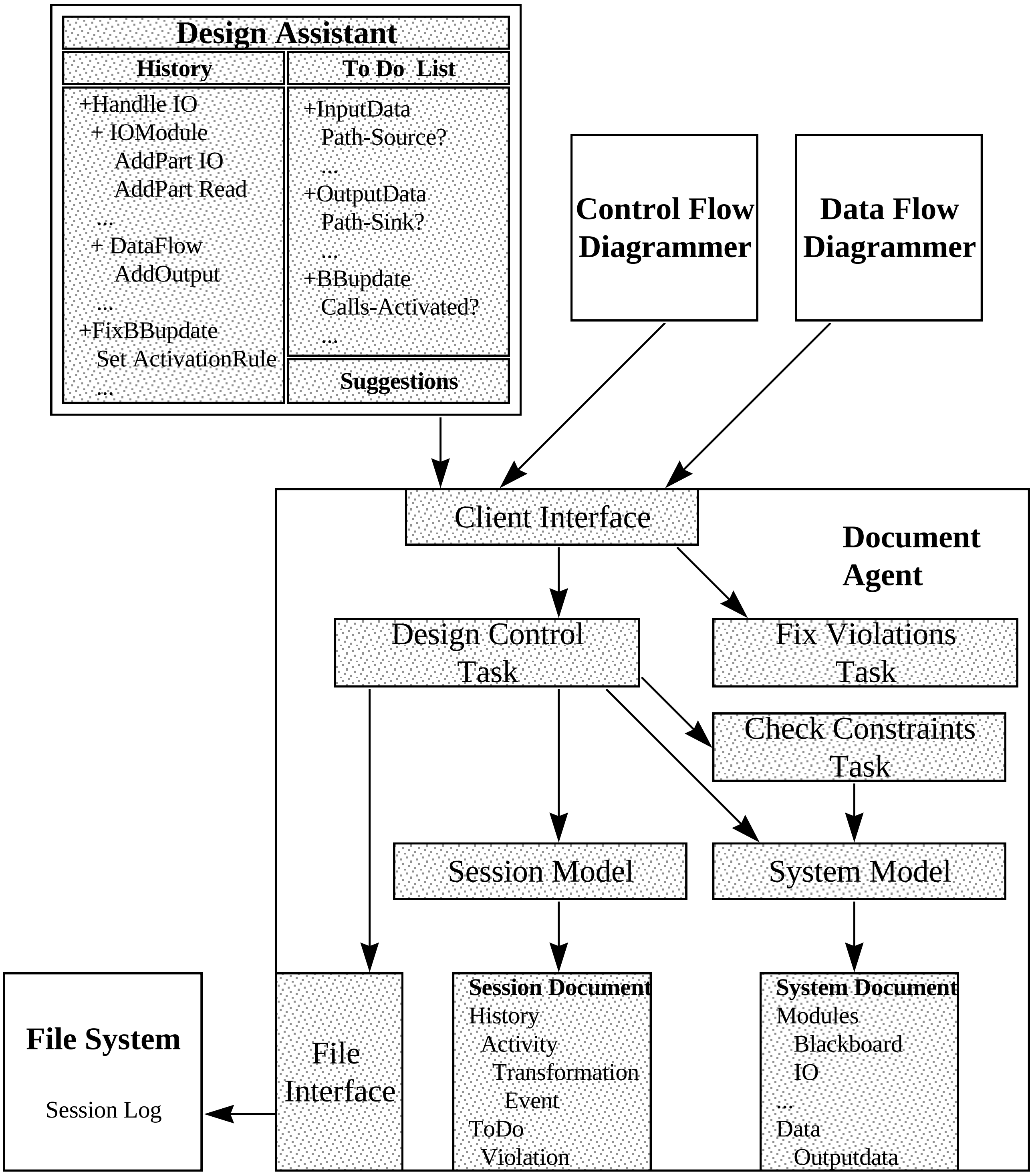


Figure 14. Document Agent for Interactive Design

The document agent had three tasks. The main task was the Design Control task which handled all requested design actions. It triggered events defined by the System model as required for a design action. Using the Session model, it recorded the resulting transformations within an activity, and the primitive events that occurred. It also interacted with a Check Constraints task. As events occurred, the Check Constraints task was notified so that it could perform actions that checked constraints. If a constraint violation occured or were resolved, then it invoked actions in the Design Control task which updated the list of Violation objects on the ToDo list in the Session Document. The Fix Violations task handled requests to provide suggestions on how to resolve constraint violations. A more detailed discussion of the Check Constraints task and the Fix Violations task is presented in section 4.4.1.

3.4.2 Document Replay

The Design Control task was also responsible for the persistent storage of session and system documents. This was achieved through a simple interface with the file system. SoDA only supported work on a single document at a time by a single designer. At any given time, there was a single design document and a single session document active. The Design Control task supported undo fully. If a system failure occurred, then recovery was only possible back to the last save of the system design. However, it was only necessary to save and store the design session document as a Session Log. The system design was retrieved from persistent storage and recreated by replaying the Session Log.

When a document is saved to persistent storage, its objects, values, and links are disassembled into some storage representation. When the document is retrieved from persistent storage, it is necessary to recreate it by assembling its objects, values and links from their storage representation. In other words, retrieving a document from persistent storage requires that the document agent perform a series of actions to recreate the design. It is often the case, that a more compact and efficient interaction with persistent storage is realized if the storage representation of a document is a log of the actions needed to recreate it. Since SoDA was already using the Session document to log the actions that created the System document, it was easier to only store the Session document.

In storing a design in this manner, we can distinguish between a compact save and a quick save. A quick save would simply dump the entire Session document. A compact save would not save the ToDo or Violation objects, since these would be recreated when the Session Log was replayed. One could also restrict oneself to simply saving the primitive Event objects that would recreate the system design, if the full historical record was no longer needed.

There were two additional advantages of this approach. SoDA was configured so that constraints and rules to check constraints could be added incrementally to the system. When a Session document was retrieved and the System document recreated, the Check Constraints task would use the new constraints and rules to check violations during the replay of the Session log. This made it possible to experiment with constraints, using a smaller less restrictive set of constraints in the initial stages of design and a more complete and restrictive set of constraints later. Storing a document as a log of actions requests to recreate it, also simplified import and export with other systems developed in the KASE project. For other systems to export a design document to SoDA, it was not necessary for them to be compatible with SoDA’s internal representation. They simply logged the same action requests defined in SoDA’s client interface.

The HandleIO activity in Figure 14, is a subset of a design session that took place when a high level design of a BlackBoard system was imported into SoDA. It was considered complete and consistent by experts associated with the KASE project. Thirty constraint violations were flagged, resulting in substantial additions and changes to the design, including special handling for input and output. Guided by SoDA, these were made in less than two hours.

3.4.3 Multiple Agent Architecture for Work Group Development

The Axiant Prototype created an electronic domain in which groups of developers could work together to build data base systems. In such a domain, the state of work is reflected in electronic documents that developers are creating to define the database system. The development of the database system is achieved by transformations of those documents.

Figure 15, diagrams the document agent architecture that was implemented to create the electronic domain. Each developer has their own computer, and each computer has its own document agent. There is a central repository for the persistent storage of documents which is accessible through a local area network. In addition, each document agent has its own workspace on the hard disk of the computer that can also be used for the persistent storage of documents. Typically, a meaningful transformation of state by a single developer can take place over the course of several days. During that time, versions of each document affected by the transformation are stored in the document workspace, leaving the versions in the document repository unaffected and accessible by other developers. There is also a remote server machine where the finished database system is installed that can be accessed by each document agent in order to compile the system or create the databases used by the system. Finally, each developer interacts with the document agent on their machine through a variety of graphical user interfaces which display documents to the developer and enact transformations by sending requests to the document agent.

Interfaces

There were three interfaces implemented within the document agent: a file interface to interact with the repository and workspace, a remote interface to interact with the compilers and databases on the remote server, and a client interface to interact with the various graphical user interfaces.

The file interface consisted of a set of simple calls to the operating system to read, write, create, copy, and delete ASCII files directly to the file system. The implementation of a LAN is such that there is no difference in the calls made to the workspace which existed locally and the calls made to the repository which existed on another machine.

The remote interface delivered asynchronous calls to a process running on the server machine which would run compiles and generate databases. The results were received by the remote interface as asynchronous responses. The document repository would be accessible by the process which ran on the server machine.

The client interface consisted of a complete set of function calls to access all the actions available in tasks implemented by the document agent. Several graphical user interfaces could be active at any one time makes requests to the document agent through this interface. As well, the graphical user interfaces could also register to be notified whenever events occurred in the document agent in order to update their displays.

Tasks

There were five tasks implemented within the document agent: a task to enact transformations of documents defined by the various document models, a task to automatically create a default form or report program, a task to compile the documents that would create a database application, a task to create a database as defined by table and domain documents, and a task to manage the state of development created by these other tasks.

The document transformation task provided actions that triggered events defined in the document models for the document agent. These actions were callable by any of the other tasks, and by any of the graphical user interfaces. It also coordinated and grouped the triggering of these events into transformations. All tasks and graphical user interfaces could define simultaneous, nested composite transformations. These were recorded to the document workspace to support recovery should the operations of the document agent be interrupted.

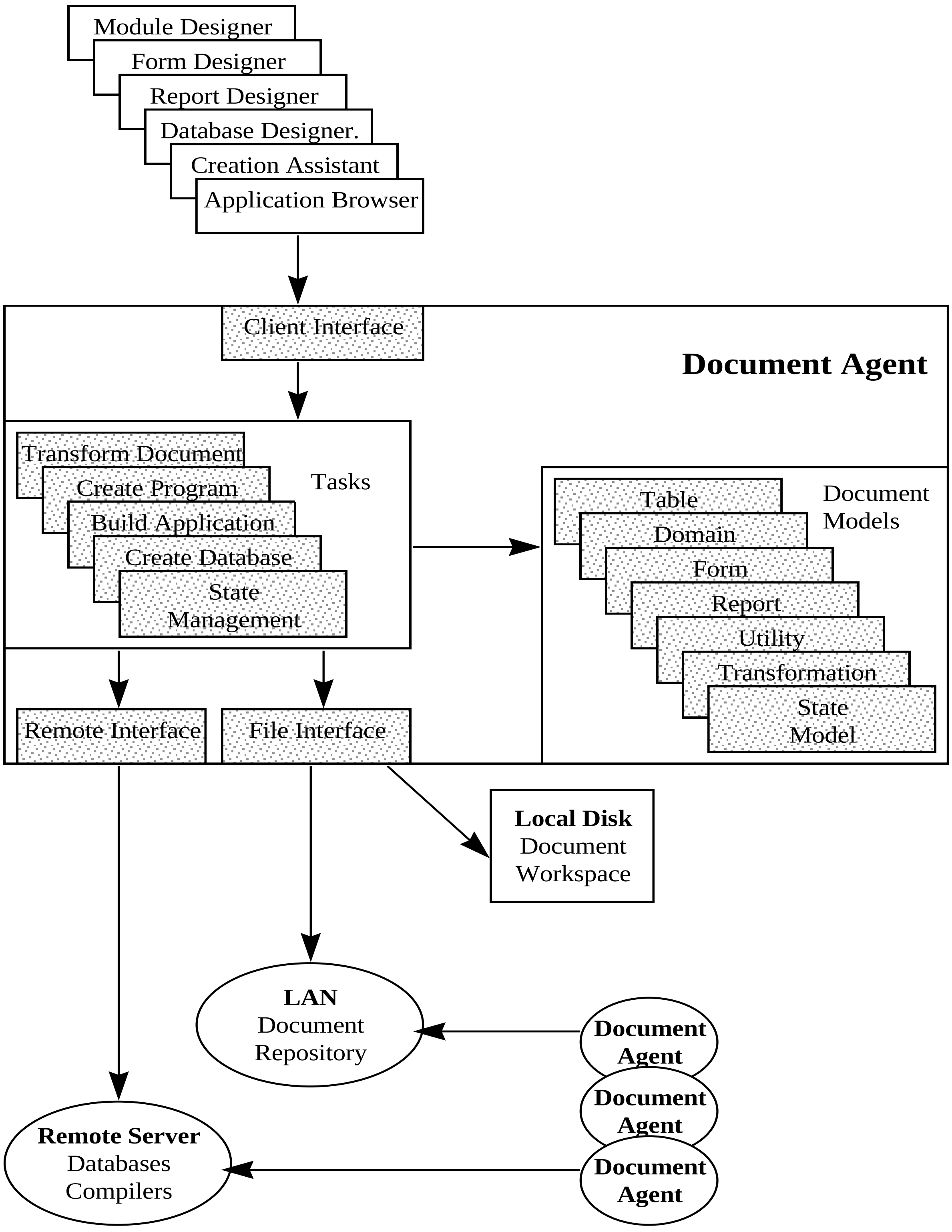


Figure 15. Document Agents for Software Development

The program creation task provided actions that were requested by the creation assistant user interface. These actions generated a ready to execute form or report program automatically based on information collected by the creation assistant. This will be described in more detail in chapter 4.4.4.

The build application task provided actions that were requested by the application browser. It determined what documents needed to be compiled in order to build an application by calling actions in the state management task. It then sent requests through the remote interface to invoke the compilation of those documents on the remote server. The task interpreted the responses received to the remote server into a form that was returned to the application browser. The database creation task interacted with the application browser and the remote server in a similar fashion.

The state management task organized the state of database system development into databases and applications. It maintained and coordinated that state in the document repository and its document workspace. Access to this state was coordinated by accounts associated with each developer. Its actions were requested by the application browser as well as the document transformation task. A full description of it is outlined in section 3.4.4

Document Models

There were several document models implemented in the document agent: table and domain documents to define records and fields in a database; form, report and utility documents to define the programs used in a database application; a transformation model used to create the session log; and a state model used by the state transformation task.

3.4.4 Configuration and Portability

The document agents were implemented in Windows. Each task and document model was a dynamically linked library that could be replaced dynamically. The document agent was also architected so that any number of these libraries could be added dynamically. At startup, the document agent read a configuration file which listed the libraries it would use for that execution. The document agent, itself was also implemented as a dynamically linked library.

Each document model implemented only the fundamental primitive actions implied in its model plus one or two special compound actions which were considered fundamental (deep copy and delete described in section 2.4.4). Additional behavior was incorporated into the domain, either by building tasks like the Create Program Task which defined transformations in terms of these primitive actions, or by defining actions in the various user interfaces which created transformations by requesting actions by the Transform Document Task.

This configuration proved very effective for the evolution of the document agent. Changes could be made to document models and tasks without requiring a time consuming rebuild of the entire system. If a new version of a document model created problems, the previous version could be substituted back in and the document agent restarted immediately.

The document agent was also robust in terms of its portability. Early versions of the document agent were implemented in UNIX and OS/2 before the Windows version. In each case, document models were reimplemented automatically by regenerating them from their model definitions (described in section 2.4.4). In a similar fashion, earlier versions of the document agent used different mechanisms for persistent storage. A relational database server, and an object database server were used via a remote interface. In both cases, the changes required were isolated in the state management task and were accomplished in less than a week.

The mechanism finally chosen for persistent storage reflected the need to organize program objects into a document format supported by third party tools. The initial efforts for persistent storage focused on the graphical objects in the user interface which were used to build database systems. It was thought that the optimum mechanism for storing objects would be to map them to a relational database. This would provide a robust, well-established storage mechanism and provide flexible support for users to do ad hoc reporting and management.

This approach was abandoned for a number of reasons. First, software development required support for managing and coordinating complex transformations of documents (not just objects) by teams of developers. Relational databases provide very little support for this type of interaction. Second, a standardized textual representation of the database systems built in terms of documents was needed for legal and practical reasons. In order, to protect their investment in database system development, it was common for organizations to require a source code listing of any database system they built. There were also a wide variety of third-party tools for configuration management and project management of software development tools which were in common usage. The Axiant Prototype either had to duplicate their sophistication, or support a storage format common to all of them. For that reason, the storage mechanism that was finally chosen for Axiant, mapped documents to an ASCII file format and provided hooks to commonly used third-party configuration management systems for version and source code control.

3.4.5 A State Management Task for Work Group Development

Figure 16 illustrates the manner in which state was managed in a multiple agent architecture. Developers could access the state of development through a browser which groups documents into the applications and databases of the database system. When a database or application was selected in the left pane of the browser, the documents which defined that application or database would be listed in the right pane. In this case, we can see the list of tables and domains which make up Database1. Table2 has a star beside it, to indicate that it has been modified by the developer in a transformation that has not been completed yet. The developer has a different version saved in her workspace, then that which exists in the document repository. Associated with each document is information that helps organize the state of development. So, for example, we can when Table2 was modified as well as comments about Table2. We can also get a listing of all the other documents which are linked to Table2. The changes made to Table2 may impact them. The developer is in the midst of assessing that impact.

The information shown in the browser is obtained by making requests to the State Management task which keeps this information in a workspace document. The workspace document is defined by a document model and describes the contents of the document agent workspace. The Workspace document model has a component Account which has information that describes the accountability linked to developer1 in this instance. The Workspace also has a Description component that defines a Description object that is externally linked to each document contained in the Workspace. Those Description objects are organized either as parts of a Database object or parts of an Application object. The external links to and from a document in the workspace are tracked as parts of the Description object for that document. This enables the State Management task to relay the information about which documents Table2 is referenced by.

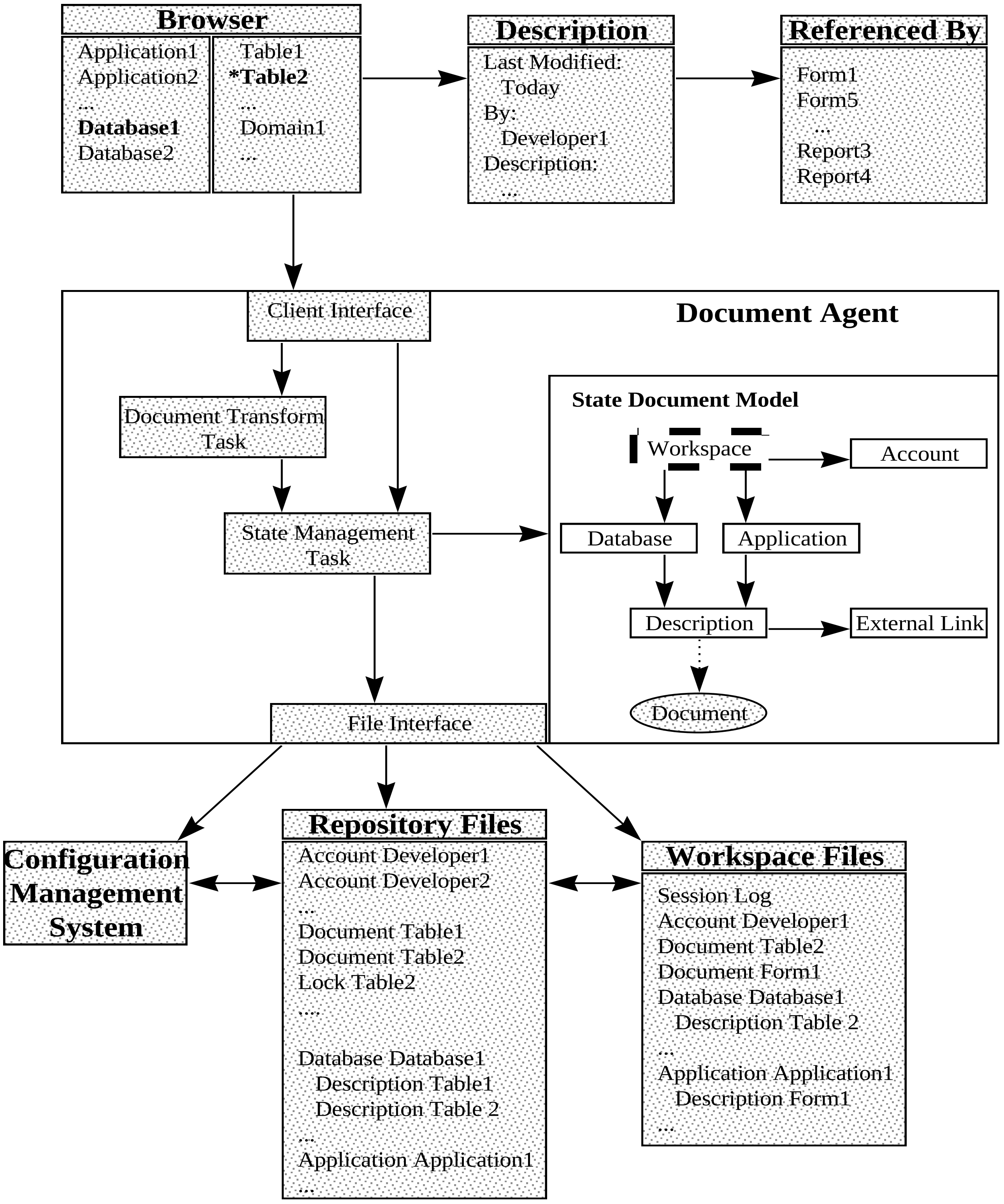


Figure 16. State Management

The State Management task locates documents in the workspace as requested by the Transform Document task or the various clients. It verifies authority to change those documents through the same interface. If a document is not currently active, the state management task will look for it in its workspace. If it is not in the workspace, then it will look for it in the document repository and load it from there. If the Transform Document task receives a request that will change a document, then the State Management task is asked to validate the request. The State Management Task will first check the current account. If the account has authority, then it will attempt to create a new version of the document in the Workspace. This is done by writing a lock file to the document repository. If a lock file already exists, then another developer is already changing the file and the request will be rejected. The Transform Document task will notify the requesting user interface with a message. If the State Management Task is able to write the lock file, then the document is copied to the document workspace and the change can be made.

The information used by the State Management task is written to persistent storage as a number of files. The Document Workspace contains the Session Log file that the Document Transformation Task saves and loads. There is also one file for the Account. The Document Repository contains one such file for each developer. When Developer1 logs in with the Document Agent, the State Management task copies the file from the Document Repository to the Document WorkSpace. Each document active in the Workspace is stored as a file which is copied from the Repository. There is a directory for each database and application that contains a document active in the database. Within those directories, is a description document for each active document. When documents are copied to the Workspace, their description files are copied as well. Any documents which are being changed are marked by a lock file written to the Document Repository.

When a transformation is complete, all files modified as a result are copied back to the Repository, and the lock files are removed. At the request of the Transform Document task, the State Management task can be asked to synchronize with the Repository. All documents which are changed in the Repository, will be into memory, replacing the old versions which existed.

State Management also provides hooks to link in a third party configuration management tool for more sophisticated version control. Whenever, a lock is successfully written and a file is about to be copied over, commands can be triggered which lock the file in the third party configuration management tool. When the transformation is complete, and files are copied over, commands can be triggered which check in the new version of the file into the third party configuration management tool.

The State Management task can also be configured to support a synchronization scheme that does not prevent conflicts from occurring. In this scenario, developers may have machines that are laptops and may want to take them away from the office to work for days or weeks. The State Management task will copy all files over that may be of use to the developer, locking the ones that the developer knows will be modified. While working the developer may decide to modify files that were not locked. In that case, a warning will be issued since the State Management task is unable to lock the file. Similarly, the locks put in place by the developer may be removed by a manager during the developers absence if it is necessary.

When the developer returns to the office, any transformations that his document agent have performed will be synchronized with the Repository using a process of two phase commitment. In the first phase, the State Management Task will attempt to establish locks for each of the documents modified by the transformation and verify that they have not been changed while the developer was away. If all the locks succeed, then there are no conflicts and the task can copy all the files back to the Repository. If any lock fails, then the transformation can not be synchronized. The developer must resolve the file(s) in conflicts. First the locks must be removed by the individuals whose accounts created the locks (or a manager). Then, if new versions of documents exist in the repository, the developer must load the new version and apply the desired changes to that version. When this is done for all files, then the transformation can be committed to the Repository.

The major lesson learned from this work is that managing the state of knowledge work captured in documents requires understanding and integration with the organizational context in which work takes place.

3.5 Relationship To Other Work

3.5.1 Research

The separation of interfaces, tasks, and document models in our architecture is analogous to architectural approaches that have been described in work on object oriented design. A control-model-view architecture has been a standard architecture for Smalltalk programs which support a graphical display [Goldberg 84]. [Jacobsen et al 92] distinguished between control, entity, and interface objects as well. The JSD method [Jackson 83] defines an architectural approach in which a domain model is created first, and then functionality is built on top of that model. [Mathiesen et al 95] extends this with an interface layer for graphical displays and integrates the object-oriented notation defined in [Coad & Yourdon 90]. We have approached our architecture for automation of knowledge work in a similar fashion. Our approach models the information in a domain of knowledge work by defining document models. The functionality or behavior that can occur in that domain is defined by tasks, while the interaction with the system is defined by interfaces.

The ability to add and replace modules within a document agent is a feature that is also present in work on Integrated Project Support Environments [Brown 91] and Portable Common Tool Environments [Campbell 88] where modules for different aspects of project support can be integrated. The Cornell Program Synthesizer [Reps 84]generated a software development environment based on grammar models of software documents. Muir [Winograd 86] provided similar software development environment where grammar models could be added and changed dynamically. [Normark 87] provided a discussion of transformations of program documents within such an environment. In the Chi system [Phillips 84], grammar-based domain models were combined with program transformations which implemented formal specifications while maintaining input and output constraints.

Our treatment of state management and transformation control addresses issues that have been handled in a different context by work on database management [Martin 89] including issues related to database transactions.

3.5.2 Industrial Practice

The concept of document agent is not commonly used or applied to systems or products that are built in industry. Nonetheless, there is an emerging technology which can be understood in these terms. Word Processors, Spreadsheets, and Drawing programs can be regarded as document agents which support work with exactly one type of document. In the case, of Word Processors there are efforts through the use of style sheets and standards initiative like SGML [Goldfarb & Rubinsky 90] [SGML 88] to extend Word Processors to support many document models.

The CALS initiative [Walter 92] is an example of creating a multiple document agent architecture with document models standardized in SGML for a particular domain of work (the creation of technical manuals on US Department of Defense contracts). In the telecommunications industry, a similar initiative is underway [Aidarous & Plevyak 94] [CCITT 92a] to standardize management of telecommunication networks. Agents and servers are defined within that architecture are defined whose operations are based on standardized object models and communication protocols and brokers [Stallings 93] [CCITT 92b].

There is also a growing number of products that provide a graphical environment for building database systems, similar to Cognos's Axiant. These include products like Microsoft's Visual Basic and PowerSoft's PowerBuilder. Lotus Notes has been one of the first commercial products to shift the emphasis in large organizations from data-based environments to document-based environments for knowledge work. It provides a flexible environment for defining and organizing information in 'notes' which are similar to our concept of document. It has also served as a platform for workflow automation which concentrates on automating the flow of information in documents as participants perform tasks.

4.0 Task Templates

Once an electronic domain has been created in which knowledge work can take place, further levels of automation can be achieved by reusing templates that characterize commonly occurring tasks. The intent is to capture the information exchange solutions created by such tasks in templates, so that they can be recreated automatically when similar situations occur. In this chapter we establish a framework for defining task templates and we discuss automation issues which arise when knowledge work is structured to take advantage of task templates.

The automation that can be realized by task templates should still be understood as enhancing and extending the capabilities of participants to perform increasingly complex work. The nature of their work will evolve, however, as participants are freed from the details of routine tasks and emphasis is placed on matching problems to known solutions. Insight is required to identify patterns that can be captured in templates. Understanding is required to integrate those templates into knowledge work.

4.1 Electronic Tasks

"The Analytical Engine weaves algebraic patterns just as the Jacquard-loom weaves flowers and leaves.

...

It can do whatever we know how to order it to perform."

Lady A.A. Lovelace, Notes upon the Memoir "Sketch of the Analytical Engine Invented by Charles Babbage", By L.F. Menabrea (Geneva, 1842), reprinted in P. and E. Morrison, *Charles Babbage and his Calculating Engines*, pp 248-9.

4.1.1 Vision

It is a premise of this thesis that knowledge work is accomplished by performing tasks which transform the state of documents. In an electronic domain, the result of such a transformation is an exchange of information that conveys knowledge for a domain of knowledge work. An electronic task, which we define as *the embodiment of a task in software*, automates work because it creates *an information exchange solution* that achieves results in the domain of knowledge work.

An electronic task embodies a task within an electronic domain by grouping actions that trigger information exchanges in electronic documents. When a participant requests an action, those information exchanges result in a state transformation for a domain of knowledge work. One can define *a solution space* for such a task by pairing the set of all document states in which an action can be requested with the set of all document states which can be created by a state transformation. Participants in a domain of knowledge work can request actions which step through the solution space of the task until the particular state they require is achieved. The result of such an interaction is a composite transformation that creates an information exchange solution on behalf of the participants.

If we call the state required by participants *a goal state*, then the creation of an information exchange solution can be characterized as a problem of creating a composite transformation that will create a goal state from an initial state. One of the classical approaches to solving any problem is to decompose the problem into parts with known solutions. We will define *a task template* as a representation of an information exchange solution. When known solutions are available in the form of task templates, tasks are performed by selecting templates which can be applied to recreate previous solutions.

4.1.2 Scenario

Let us consider a scenario which illustrates electronic tasks. A banker is putting together a portfolio proposal for a prospective new client Wilma, and her husband Fred. The banker would like to convince her to move all her banking affairs to Banks 'R Us. As shown in figure 17, the relevant information about Wilma is available in a Client Form. The products and services offered by the bank is available in a Product Catalog document. The bank has also put together a Portfolio Guidelines document to assist in the task of creating proposals like this.

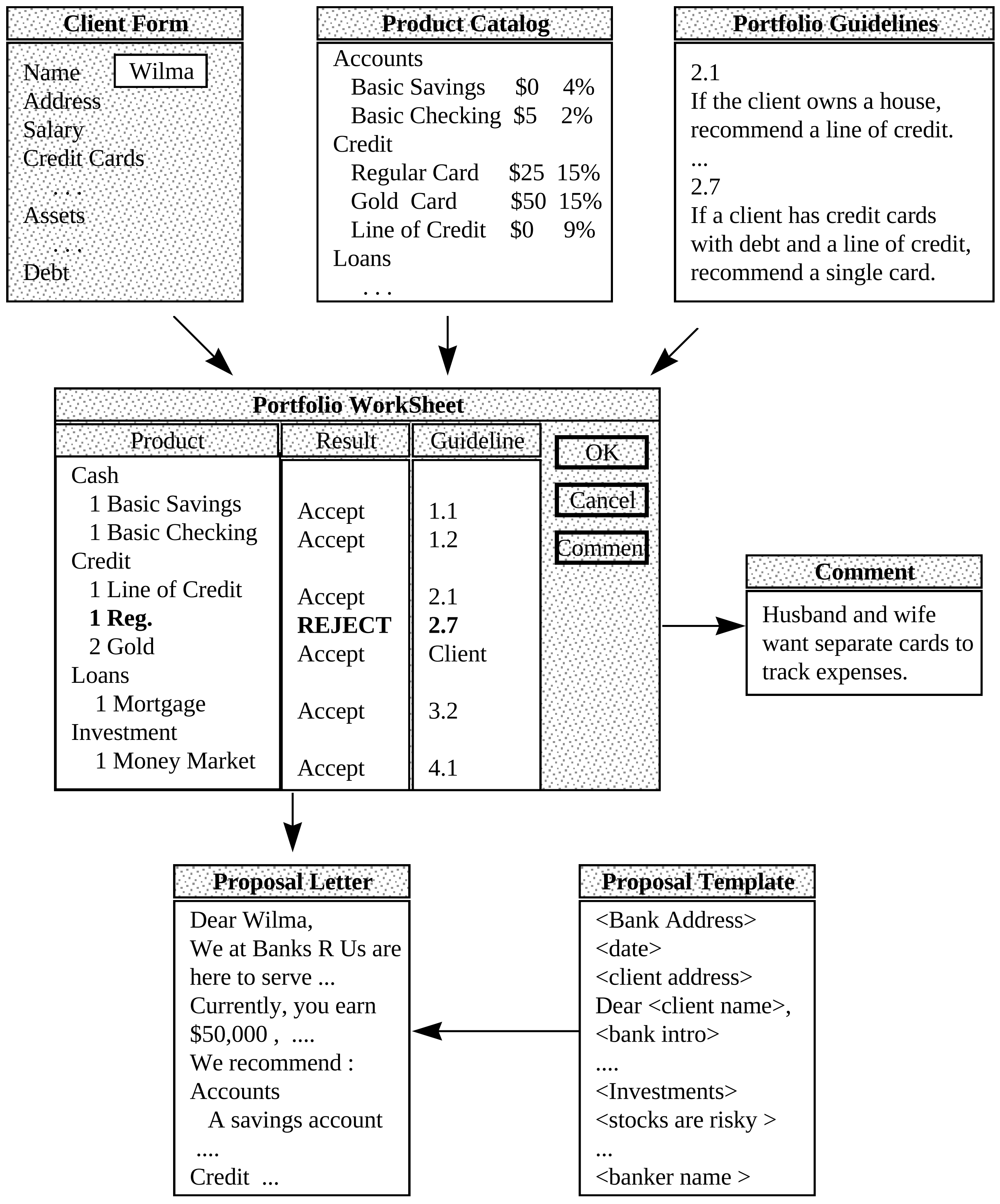
The banker creates his proposal in a Proposal Worksheet document. The banker scans through the guidelines, checking against the Client Form, to see which guidelines apply to Wilma. Whenever, a guideline applies to Wilma, the banker looks the product up in the Product Catalog and enters it in the Proposal Worksheet. Wilma owns a house, so following guideline 2.1, the banker enters a Line of Credit product in the Worksheet. Wilma also has several credit cards with debt owing, so following guideline 2.7, the banker enters 1 credit card.[7]

Having formulated the proposal in the Portfolio Worksheet, the banker would like to present the proposal to Wilma in the form of a letter. The bank has set up a standard template to be used in writing such a letter. The banker follows the template, entering information specific to Wilma: her address, her current banking information, and the portfolio the banker will be recommending. Much of the information required for the letter is already in the template: the bank's address, a standard introduction, etc. Some information is not appropriate for Wilma so the banker deletes it. There is a standard disclaimer about the risks of investing in mutual funds of stocks, but it is not needed because the banker has not recommended such funds for Wilma.

When the banker presents the proposal to Wilma, she likes the proposal and accepts it, except for one recommendation. While the idea of using a line of credit to pay off her credit cards appeals to Wilma, she prefers that Fred and her each have a separate card for their purchases so they can be tracked with separate bills. The banker agrees with this idea, and updates the Portfolio Worksheet. He marks the recommendation of one card as rejected, and enters a second recommendation for two cards. He also attaches a comment to the previous recommendation indicating the reason why Wilma rejected it. As Wilma leaves, the banker commences with his next task of filling in the order forms to enact the proposal.

The bank keeps track of recommendations in order to get feedback on how effective their proposals are in the marketplace. Whenever a recommendation is rejected, a record of the rejection and the reason for rejection is automatically forwarded to the marketing department. In fact, someone there has noticed that Wilma and Fred are not the only couple that like having two credit cards. Many of the Proposal Worksheets at the bank, have had the 1 credit card recommendation rejected for the same reason. At the next marketing meeting, a proposal will be put forward that the bank change guideline 2.7 for married couples to two cards. It will also be proposed that the bank could attract more business by making a special offer in their next marketing campaign to waive the annual fees on the second card.

[7] A line of credit is money that a client can borrow, using some asset as a guarantee that the money will be repaid. Because it is backed by an asset guarantee, the interest rates are much lower than for credit cards. Owning a house, it makes more sense for Wilma and Fred to have a single credit card that they pay off every month, using their line of credit if necessary.



Client Form

Name	Wilma
Address	
Salary	
Credit Cards	
. . .	
Assets	
. . .	
Debt	

Product Catalog

Accounts		
Basic Savings	$0	4%
Basic Checking	$5	2%
Credit		
Regular Card	$25	15%
Gold Card	$50	15%
Line of Credit	$0	9%
Loans		
. . .		

Portfolio Guidelines

2.1
If the client owns a house, recommend a line of credit.
...
2.7
If a client has credit cards with debt and a line of credit, recommend a single card.

Portfolio WorkSheet

Product	Result	Guideline
Cash		
1 Basic Savings	Accept	1.1
1 Basic Checking	Accept	1.2
Credit		
1 Line of Credit	Accept	2.1
1 Reg.	**REJECT**	**2.7**
2 Gold	Accept	Client
Loans		
1 Mortgage	Accept	3.2
Investment		
1 Money Market	Accept	4.1

Comment

Husband and wife want separate cards to track expenses.

Proposal Letter

Dear Wilma,
We at Banks R Us are here to serve ...
Currently, you earn $50,000 , ...
We recommend :
Accounts
A savings account
....
Credit ...

Proposal Template

<Bank Address>
<date>
<client address>
Dear <client name>,
<bank intro>
....
<Investments>
<stocks are risky >
...
<banker name >

Figure 17. Formulating a Proposal Scenario

If we view this scenario as the creation of an information exchange solution, then some observations can be made. The problem of creating a proposal to present to Wilma can be decomposed into steps. First, determine what products will be recommended, then write a letter presenting the proposal. The problem of determining what products to recommend, can also be broken down into steps. For each product in the catalog, determine whether or not it should be recommended. Once the problem has been decomposed into steps, there are existing solutions or *task templates* that can be used. The Product Guidelines document has *rules* for each product that determine when they should be recommended. The Letter Template is a *profile* that defines what a Proposal Letter document should look like. It creates a partial solution to the problem of writing a letter, but leaves smaller problems to solve. Certain parameters need be filled in from the Client Form, the Product Catalog, and the Portfolio Worksheet. There is also a number of optional sections that need to be written. The writing of these sections in the letter, is organized as a *list* of solutions from which the banker can simply choose.

The Portfolio Proposal task is accomplished be creating a composite transformation that includes the instantiation of task templates to provide information exchange solutions for commonly occurring steps in the overall problem[8].

[8] In fact, our characterization of the Portfolio Proposal Task in this scenario can be taken as an initial definition of a task template for the overall problem of creating Portfolio Proposals.

4.2 An Information Exchange Solution

In this section, we will illustrate how task templates can be created by structuring a task in terms of classical approaches to problem solving. Our intention is to automate the creation of information exchange solutions for commonly occurring tasks. We will characterize problem solving in a manner which is similar to the standard approach taken in artificial intelligence [Amarel 68]. Our framework for defining task templates will be analogous to approaches taken in work on expert systems [Buchannon 82]. Our emphasis, however, will be on structuring knowledge work tasks to improve the level of automated support and guidance that a document agent can provide to a participant [Nanard 93].

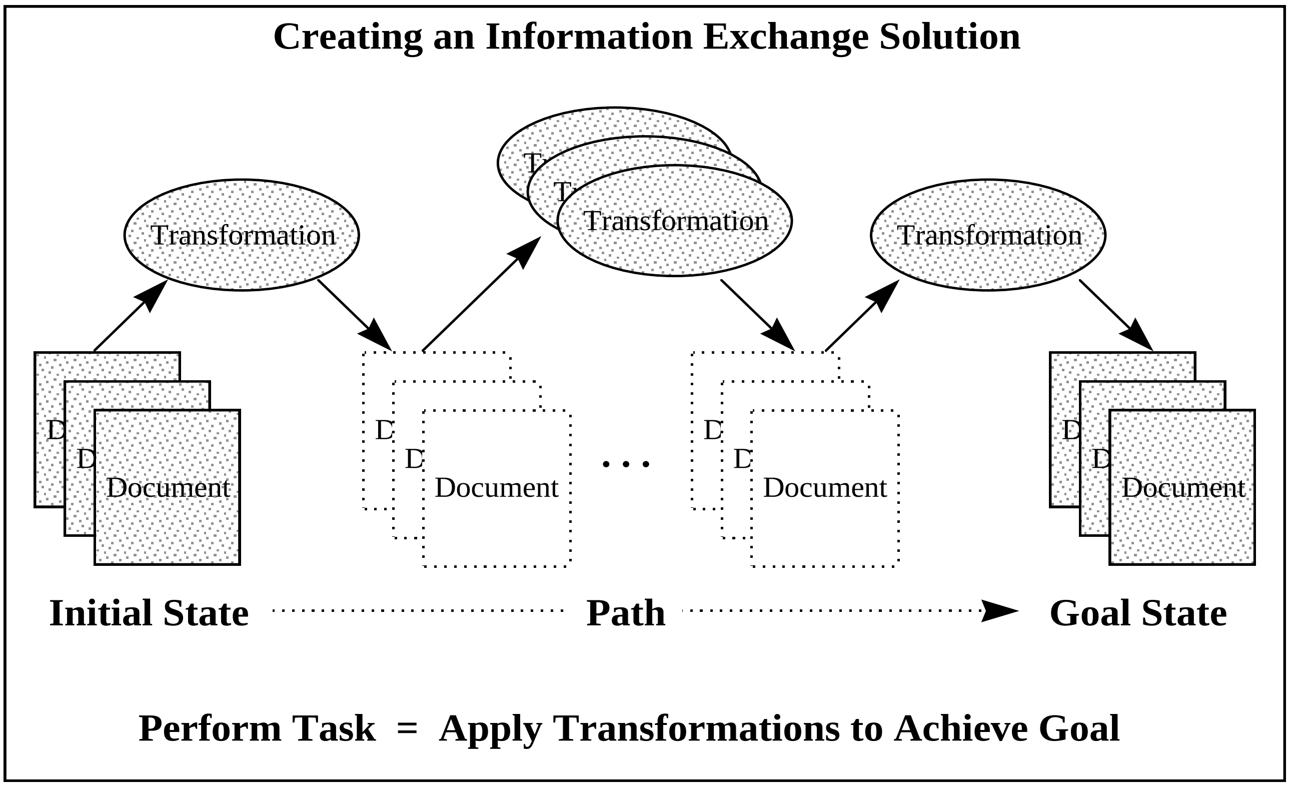
Figure 18 illustrates our approach. The top box characterizes an information exchange solution that was created by a document agent in response to requests from a participant. The initial state of the work is captured in the state of documents controlled by the document agent. The actions requested by the participant result in a sequence of transformations. Each transformation is a step that creates a new state until a final state is created. The final state is a goal state in which the completion of the task by the participant is reflected in the state of documents controlled by the document agent.

A task template characterizes an information exchange solution, or key elements of that solution, so that they can be reused to automate the creation of similar information exchange solutions. The states along the path of the solution that is created can be characterized by a state profile which describes the documents in those states. One way to reuse an information exchange solution is to simply save the documents created in the goal state. When a similar information exchange solution is needed again, those documents are copied and then modified as needed. The parts of the documents which vary from solution to solution can be captured in parameters so that modifications are created by instantiating parameters with values for a particular solution.

The transformations which created an information exchange solution can be characterized by transformation rules that define the actions which triggered the transformation. One way to reuse an information exchange solution is to simply record the actions requested in creating the goal state. A similar information exchange solution can be created by replaying the actions. Variations from the original solution can be captured in parameters associated with the actions defined by transformation rules.

The two approaches can be combined. An information exchange solution can be decomposed into subproblems. Each subproblem has an information exchange solution that can be combined to create the overall solution. Some aspects of the goal state can be created by copying documents, other aspects can be created by replaying transformations. The process of creating an overall information exchange becomes one of identifying subproblems, selecting task templates, applying them and selecting values to instantiate parameters. Decision lists can be used to characterize the choice of templates for subproblems, and the choice of values for parameters.

The analysis of task templates in this section is a representative, but not an exhaustive, presentation of the mechanisms which can be used to automate tasks. There are also some practical considerations with respect to this approach to automation which are discussed in Section 4.3.



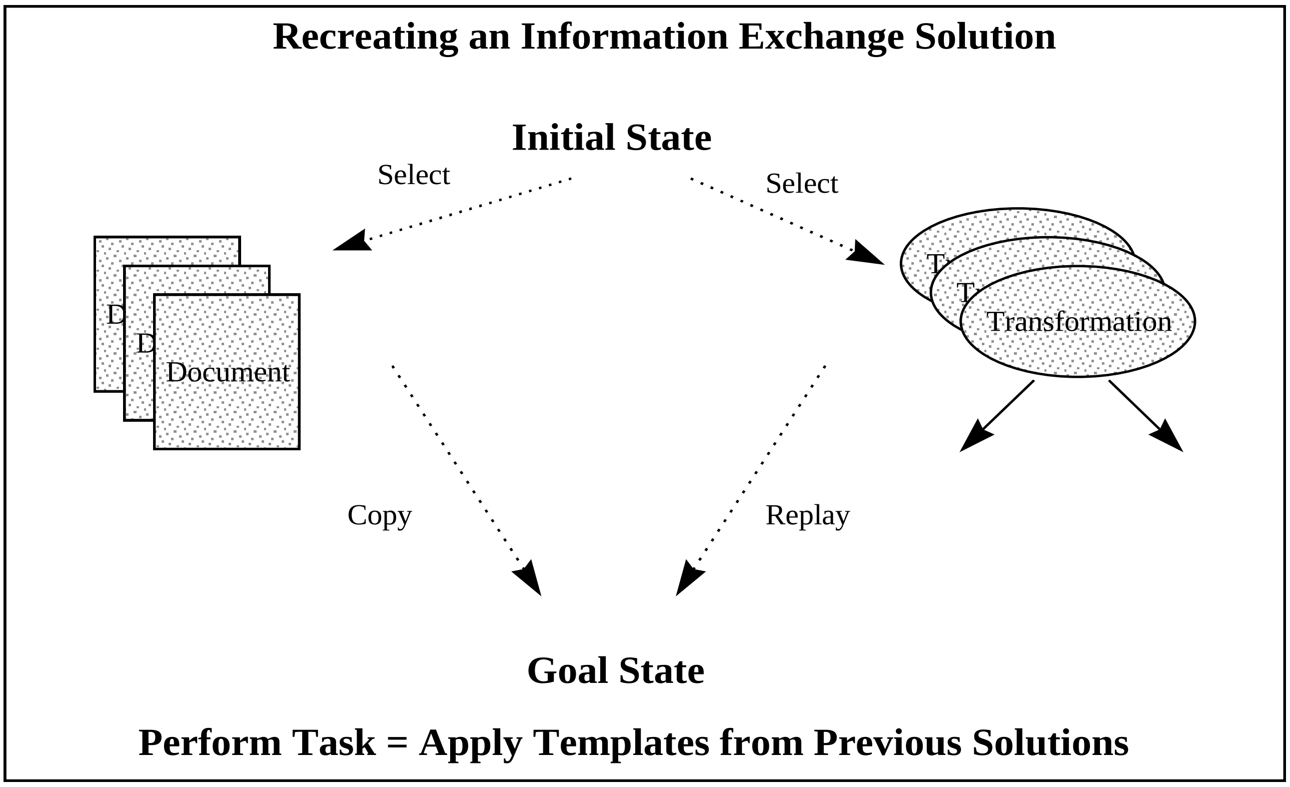

Recreating an Information Exchange Solution
Initial State
Select
Select
Document
Transformation
Copy
Replay
Goal State
Perform Task = Apply Templates from Previous Solutions

Figure 18. Understanding Task Templates

4.2.1 Task Template

A task template is a representation of an information exchange solution that can be used to generate similar solutions within a domain of information exchange.

An information exchange solution is a sequence of transformations which transform the state of a domain through a sequence of states in the solution space of a task from an initial state to a goal state. A task template represents an information exchange solution by characterizing the states and/or transformations along the path from an initial state to a goal state. It describes the initial state in which it can be applied, the path that it will follow through the solution space of a task, and the goal state that will result. A task template can be parametrized in order to extend the characterization of a particular information exchange solution to a set of similar solutions. A particular information solution is generated by copying states or replaying transformations with a particular set of parameter values. Templates can be used to guide and structure the work of participants. They can also be used within a document agent to automate the generation of information exchange solutions.

Initial State

The initial state of an information exchange solution consists of all the information and associations of all the documents and their components which existed in a domain of exchange at the moment the first transformation in the solution was triggered. A characterization of initial state need only focus on the particular information and associations of documents and components which actually participated in the information exchange. The initial state of a task template is characterized in this way by means of a *state profile*.

Goal State

The goal state of an information exchange solution consists of all the information and associations of all the documents and their components which existed in a domain of exchange at the moment the last transformation of the solution was completed. A characterization of goal state need only focus on the particular information and associations of documents and components that actually resulted from the information exchange. The goal state of a task template is characterized in this way by means of a *state profile*.

Path

The path of an information exchange solution consists of a sequence of states from the initial state to the goal state and the sequence of transformations which created each state from the preceding one. A path can be characterized by means of a sequence of *transformation rules* each of which describes a transformation and the state in which it was triggered.

Parameters

A task template can be parametrized to accommodate variances in the states and transformations of paths for similar information exchange solutions. Instantiating the parameters of a task template effectively determines the choice of initial state, goal state, and path from initial state to goal state. The instantiation of each parameter is a subproblem that is solved in order to complete the instantiation of an information exchange solution from a task template. In many cases, those subproblems can be characterized by means of *decision lists* which enumerate the possible choices for a parameter.

Example

We can illustrate a task template by defining a template for the creation of the portfolio proposal in our scenario. In that scenario a portfolio proposal was created for Wilma. Our task template is parametrized so that it can be used by any banker for any client. An initial state is characterized in terms of an active account for a banker, a client form for a client and the Product Catalog and Portfolio Guidelines for Banks ‘R Us. The goal state is characterized in terms of a Portfolio Worksheet for the client and a Proposal Letter from the banker to the client. The Path is characterized in terms of a transformation from the initial state in which a Portfolio Worksheet is created for the client, followed by a transformation from the resulting state in which the Letter Template is copied and filled in.

Portfolio Proposal **Template**

Parameters: Banker = name of a banker, Client = name of a client

Initial State: Account(accountname=Banker),
Client Form(clientname=Client),
Product Catalog(bank=“Banks ‘R Us”),
Portfolio Guidelines(bank=“Banks ‘R Us”)

Goal State: Portfolio Worksheet (clientname=Client),
Proposal Letter(accountname=Banker, clientname=Client)

Path:

```
From Initial State
        Create Portfolio Worksheet (clientname=Client)
From Initial State & Portfolio Worksheet
        Copy Letter Template into Proposal Letter
        Choose from list of optional sections
        FOR each parameter  DO enter information
```

4.2.2 State Profile

A representation of the context in which an information exchange solution is created in terms of documents, objects, values, and links.

A state profile characterizes the state of a domain of information exchange by representing the particular documents, objects, values, and links which are relevant to an information exchange solution. The context in which a solution is achieved can be defined by state profiles of the sequence of states in the path of the solution from initial state to goal state. If the context is captured in a single parametrized state profile for the entire sequence of states in the path, then instantiating the solution is reduced to instantiating the parameters of that state profile. A state profile can be implemented in the form of document models, document templates, or constraints.

Constraints

The state profile of a task template can be captured by constraints which identify properties of a document instance that must be present in order for it to belong to the context. The constraints are associated with a document model and they are defined as actions in a task. A constraint triggers events which determine whether or not a property is present. As a solution is being created, a task can use the constraints defined in the state profile to prevent document instances which do not belong to the context.

Document Templates

The state profile of a task template can also be captured in document instances that are representative examples of the types of documents which make up a state profile. These document instances define a document template for the state profile. All states in the path of the information exchange solution will be similar to this document template. An information exchange solution is created by copying the document templates which define the state profile and instantiating their parameters. Task actions can then be requested by a participant to customize the information exchange solution as needed for a particular situation.

Document Models

A state profile is a characterization of documents, objects, values, and links. A state profile can be captured in a document model or set of document models which define this characterization as a system of information exchange. The state profile is parameterized by the definition of parts, information, and associations in these models which restricts the set of possible objects, values and links. The definition of behavior in these models restricts the events, and therefore transformations, which can be triggered to instantiate the parameters of the profile. Instantiating a task template whose context is characterized by a state profile defined as a document model is accomplished by the actions of a task which performs transformations of the state profile. The actions can trigger events which instantiate the parameters of the state profile to create a goal state. Or the actions can instantiate the parameters of the state profile to create an initial state from which the transformations defined in the path of the task template can be applied.

Example

To illustrate a state profile, let us return to the Portfolio Proposal Template that we created as an example in Section 4.2.1. In the path of that template, we first created a Portfolio worksheet and then we created a Proposal Letter. The Letter Template that was copied and modified is an example of a document template. Since the banker starts by copying the Letter Template, any Proposal Letter that is created by the banker will be similar to the Letter Template. The Letter Template is a state profile that defines a context for the creation of a solution in the form of a Proposal Letter. There are parameters that need to be instantiated, but the banker can also make additional modifications to customize the letter as he sees fit.

The Letter Template can be used as the basis for a simple Proposal Letter Template. We can redefine our Portfolio Proposal Template to instantiate the Proposal Letter Template.

Portfolio Proposal **Template**

. . .

```
Path:           From Initial State
                        Create Portfolio Worksheet (clientname=Client)
                        Instantiate Proposal Letter Template
```

Proposal Letter **Template**

```
Parameters:     Banker = name of a banker, Client = name of a client

Initial State:  Proposal Letter = Copy of Letter Template

Goal State:     Proposal Letter = modified Copy of Letter Template

Path:           From Initial State
                        Choose from list of optional sections
                        FOR each parameter  DO enter information
                        Modify as needed until banker is satisfied
```

If we wanted to control and automate the Proposal Letter Template, we could use a document model instead of the Letter Template. The initial state would be an uncompleted instance defined by the Proposal Letter document model, and the goal state would be a completed instance. The path would be similar. Each transformation in the path would be defined by an action in the document model. However, the model would automatically fill in the parameters when it created a Proposal Letter instance using its links to a Banker Account, Client Form, Portfolio Worksheet, and Product Catalog. As well, the banker, in requesting modifications, would be restricted to adding text.

Proposal Letter **Document Model**

Parts: CannedText[n], OptionalText[n], CustomizedText[n],Parameter[n]

Information: Bankername, Clientname, Date

Behavior: CreateLetter(Bankername, Clientname), ChooseSections(), AddText()

Associations:

Owned: ClientParameters (Proposal Letter, Form)[1] ,
PortfolioParamameters (Proposal Letter, Worksheet)[1],
ProductParameters (Proposal Letter, Catalog)[1],
BankerParamameters (Proposal Letter, Account) [1]

4.2.3 Transformation Rule

A representation of a transformation that links one state in the path of an information exchange solution to another state in the same path.

A transformation rule characterizes a step in the path of an information solution by representing the change from one state to the next. A transformation rule can define state profiles which characterize the two states. It can also define actions that identify a start state and describe the transformation which creates the next state from the start state. A transformation rule can also define a monitor which characterizes the previous step in the path and the transformation which will create the next state from the state created by that step.

State Profiles

A transformation rule can consist of state profiles for the starting state and the resulting state. There must be a relationship between the two states which defines how to create one from the other. If the state profiles are defined by a document model or a document template, then the profile for each state will be a document instance. A transformation rule could simply link the two document instances, so that whenever the first instance appeared, an instantiation of the transformation rule would simple replace the first instance with the second. If the state profile is parametrized, then the instantiation of parameters in the second state can be defined in terms of the values used to instantiate the parameters in the first state.

The state profiles may also be defined by constraints which express the properties that are true in each state. Whenever a document instance exists in which the constraints defined by the state profile for the first state are true, an instantiation of the transformation rule would require that the next step in creating a solution be accomplished by creating a transformation which results in a state where the constraints specified in the state profile for the second state are true. Either the task will incorporate an inference engine to determine the transformation required, or a participant will request actions to create the needed transformation. In either case, the transformation is not complete until it has been determined that the constraints are true.

Actions

A transformation rule can consist of two actions, one for the each state. The first action is a constraint action which identifies the starting state for this step in the information solution. The constraint action identifies properties of a document instance which must be present in order for it to be a starting state. The second action defines a transformation which will create the second state from the first state. A task instantiates a transformation rule defined in this manner by triggering the constraint action. If it returns a value of true then the second action is triggered. There may be inputs to the first action which must be instantiated by the task. The second task may have inputs to be instantiated as well. The inputs and outputs of the first task may be used for this purpose.

Monitors

A transformation rule can consist of a monitor which associates a transformation with a previous step in the path of a solution. The previous step is defined by the description of the event triggered by that step. The state which results from that previous step is now the start state for the step defined by the monitor. The transformation for that step is defined by an action. In order to instantiate a monitor, a task will need to receive notification when the event occurs. A monitor is instantiated by triggering the action whenever an event occurs that matches the monitors descriptions. The inputs and outputs of the previous event may be used to instantiate the inputs of the action.

Example

We can illustrate transformation rules, by considering the rules specified in the Portfolio Guidelines document of our scenario. The rule from section 2.1 is clearly defined in terms of actions:

Rule 2.1 If the client owns a house, recommend a line of credit.

The banker must check that the property of owning a house is true about the client. If it is true, then the banker should perform the action of entering a line of credit product in the Portfolio Worksheet. A very simple task template can be defined around such a rule. In fact, each rule in the Portfolios Guidelines document is a template for a solution as to whether or not a product should be recommended. We can rewrite the path description of our Portfolio Proposal template to reuse these solutions.

Portfolio Proposal **Template**

. . .

```
Path:           From Initial State
                          Instantiate each Template in the Portfolio Guidelines document
                          Instantiate Proposal Letter Template
```

Line of Credit **Template**

Parameters: Client = name of a client

Initial State: Client Form(clientname=Client), Portfolio Worksheet (clientname=Client)

Goal State: Client Form(clientname=Client),
Portfolio Worksheet (clientname=Client, Line of Credit Recommendation)

```
Path:           From Initial State do
                          trigger Constraint Action of Rule 2.1
                          If Output=True THEN DO
                                    trigger Transformation Action of Rule 2.1
```

If the bank had not formulated a Portfolios Guidelines document, but it did have a large collection of previously successful proposals in a Portfolios History document, then a different set of transformation rules could be used. Each successful proposal could be defined as a transformation rule that linked a particular Client Form with a successful Portfolio Worksheet. In order to create a Portfolio Worksheet for Wilma, a task would attempt to instantiate each transformation rule defined by a previously successful proposal, until if found one whose Client Form matched Wilma's. Then, the Portfolio Worksheet for that client would be copied to create one for Wilma. The task, of course, would have to create a Compare Action that could determine whether or not two Client Forms should be considered a "match".

Successful Client #343 **Template**

Initial State: Client Form(clientname=Client, ... assettype="House" ....)

Goal State: Portfolio Worksheet (clientname=Client, ... product="Line of Credit" ....)

4.2.4 Decision List

A representation of a choice of information exchange solutions that can be reused in the creation of a new information exchange solution.

A decision list characterizes the choices to be made in creating a new information exchange solution from existing solutions represented by task templates. In a particular situation, there may be several templates which are applicable. Once a template is chosen, there may be parameters to instantiate. For each parameter, there will usually be a range of possible values to choose from. A choice is a subproblem associated with the instantiation of a task template whose possible solutions can be represented by a decision list. That list enumerates, filters, and orders all the possible solutions so that the only action required is to select. Depending on the situation a decision list can be classified as mutually exclusive (only one selection can be chosen), or exhaustive (any number of selections can be chosen).

Template Selections

For a particular type of task problem, there may be many possible solutions which are applicable. As such, there may be many templates to choose from. Identifying the possible templates for a given situation, and organizing them in an order that is useful can assist in the decision of which solution to instantiate. A particular task problem may also be approached by decomposing it into subproblems for which known solutions exist. A task template can be created for such a task problem, in which each step of the path solves a subproblem by instantiating a task template. A decision list can identify, organize, and order the subproblems into a sequence that defines the path of the task template. For each subproblem, another decision list can identify the possible templates to instantiate for the subproblem.

Parameter Assignments

When a template is instantiated there are parameters to instantiate. These parameters are effectively part of the context for a task template. They will be instantiate or assigned documents, objects, values or links which exist in the initial state of the template. The possible assignments for a parameter are defined by the document model which describes the document, object, value or link. In many cases, the theoretical range described by the document model will be quite large. A decision list can consist of a practical list of predefined instances which the template will be restricted to. It can also consist solely of instances which are in existence in the state at the moment a task template is instantiated. The list can also be filtered and sorted according to constraints specified for the state profile of a task template. Characterized in this manner, instantiating a parameter is solved by selecting an assignment or assignments from a decision list.

Example

We can illustrate decision lists, by examining the Portfolio Proposal task template we created in the example at the end of section 4.3. In particular, consider the second step in the path which is now described by a line that says “Instantiate Proposal Letter Template”. It might be useful to have several templates to choose from: one for married couples, one for individuals in a certain income bracket, etc. A decision list could organize and filter the list of possible templates, either automatically instantiating the template that appears first in the list, or presenting it as a choice for the banker.

Once the letter template is instantiated, one of the actions required is to choose the optional sections to include. We have not described it as such, but the list of optional sections is in reality a parameter of the Proposal Letter Template. All possible optional sections have been identified and they can be organized into a decision list. That decision list can also be filtered depending on the particular products recommended for a client (e.g. stocks) and the specific information which describes a client (e.g. married or not married). Client Form being used.

A similar analysis also applies to the first step of the path which is now described by a line that says “Instantiate each Template in the Portfolio Guidelines document”. This is really a sequence of steps which resulted from us decomposing the problem of creating a Portfolio Worksheet into many small problems of deciding whether or not to recommend a product. In essence, the Portfolio Guidelines document has now become a decision list which identifies all the subproblems in creating a Portfolio Worksheet, associates a template with each subproblem, and orders those templates. It is important that the template for rule 2.7 “Recommend 1 credit card if there is debt and a Line of Credit has been recommended” must be instantiated after rule 2.1 which determines whether or not a Line of Credit should be recommended.

Our presentation has also simplified the complexity in decomposing a problem in this fashion. If rule 2.7 recommends 1 card, and another rule recommends 2 cards, what should be recommended. A more sophisticated decision list would create sublists that organize the rule templates into groups of mutually exclusive templates. The 1 card rule, would be in the same group as the 2 card rule, and the decision list for that group would organize the templates for each rule in such a fashion that only would could be used.

A predefined list of choices for information entered in the Client Form and other documents is important to enable the automation associated with templates. Rule 2.1 states that if there is a House asset recommend a line of credit. What if the asset is a Townhouse, or a Cottage, or a Dwelling , etc.? Predefined lists of choices for values in the state profile standardize the information collected and used in each of the different steps of the template.

4.3 Work Automation

We have defined a task template as an information exchange solution. Our intention in defining task templates in this manner, is that the performance of a task should be structured to reuse existing solutions whenever a similar problem is encountered. Those solutions should also be reused whenever more complex problems are encountered, by decomposing the complex problem into smaller problems that are similar to existing solutions in the domain. This is a natural approach to solving problems in any form. When solutions are represented as task templates, and the tasks inside a document agent are implemented to use these task templates, then we have a powerful mechanism for automating tasks within a domain of knowledge work.

There are a number of issues which arise when one attempts to automate work in this manner. One set of issues is concerned with the manner in which a task should be implemented within a document agent in order to maximize both flexibility and automation in solving problems with task templates. Another set of issues is concerned with determining the degree of standardization that is required and appropriate in order to achieve automation of tasks within a domain of knowledge work.

4.3.1 Task Implementation

Different approaches can be taken to implementing a task so that it uses task templates. At one extreme, a task can simply provide mechanisms for defining and accessing task templates which participants use to structure their work. Participants can group representative documents for reuse, and the document agent can provide a decision list to choose one as a starting point for document creation, whenever a new document is opened. A scripting or macro facility can also be provided which allows a participant to record or define sequences of actions that are commonly performed. One can view such a macro as a transformation rule which is linked to the previous step in a solution, whenever the event occurs that is associated with a request for that macro. A task can also provide a facility for creating and linking to electronic text that provides rules and guidelines for the performance of a task as well as mechanisms for linking to the appropriate sections of that text while the task is being performed. The first two facilities have become quite common in commercial products that can be viewed as document agents, but the third is usually only available in a pre-defined form that explains the pre-defined actions built into the document agent.

At another extreme, a task can be completely automated in its used of task templates. Either the task is triggered automatically, or a single request is required in order to create a solution. In a complex situation, such a task may have to reason or search within its solution space in order to construct a solution from the pre-defined solutions it has available as task templates. Such as system may be referred to as an expert system. There is a large body of literature which describes how expert systems can be built and there is commercial software available to assist in such an endeavor. In a less complex situation, an algorithm for constructing a solution may be known and the construction of a complex transformation from task templates can be defined by a procedure.

Even in the case of complete automation, the ability of participants to configure the behavior of the task is important. In an expert system, the templates which drive the operation of the task are defined by participants and must be available in a form to justify or verify the task’s behavior. In a procedural case, the behavior of a task can be tailored by decisions lists associated with state profiles from which parameters can be preset. An example of this might be the manner in which a complex collection of user preferences and default settings can be preset for the task of printing a document.

4.3.2 Degree of Standardization

In considering the division of labor between participants and a document agent it is important to bear in mind that automation is based on standardization. A task template is not only a solution that can be reused, it is the identification and standardization of a commonly occurring pattern that ensures and imposes a conformity in the performance of a task. Automation is possible when the focus of the pattern is narrow enough that the essential aspects of state and transformation can be predefined completely and precisely. Such a definition can be time consuming and costly. It may also be equally time consuming and costly to change or adapt that definition if the task is changing constantly (for example, if the bank is constantly changing its Product Catalog so that the rules in the Portfolio Guidelines are no longer applicable.)

It is often not practical to automate tasks where standardization is difficult or inappropriate. If a task is rarely performed, or if it can already be performed quickly with little apparent effort by people then there may be very little benefit gained from automation. Often a simple decision list of representative examples that participants can choose from as a starting point for a task, or as a guide for comparison is enough to provide the major part of any benefits that can be obtained. It has been our experience, that Pareto's Principle[9] is relevant: about eighty per cent of total automation is achieved by about twenty per cent of the effort that would be required to create total automation. Cost benefits analysis should probably be considered before automation beyond eighty per cent is undertaken.

There may be situations where standardization is appropriate but difficult. If a task is performed often enough, or the performance of the task by people is considered time-consuming and arduous, then the benefits gained may justify large efforts. A deep understanding of the task can sometimes simplify those efforts. In some situations a partial solution can be effective by decomposing a task into subproblems some of which are automated and some of which are performed by people. Partial automation can still be effective in reducing the time taken or the difficulty associated with a task. In other situations, a task template whose instantiation only creates a suitable solution half of the time can still be effective. Intervention must be planned to recognize when a solution is not suitable and then to create a suitable one through other means.

The process of standardization can result in insights that provide benefits which go beyond the gains associated directly with automation. For example, the standardization of product selection in a Portfolio Guidelines can be a major undertaking, especially if there is a dual requirement of organizing the form of the guidelines both for human communication and machine activation. The analysis of how a bank's products are matched to meet the needs of a client within a particular domain of concern can provide powerful insights into the nature of a bank's business and the source of its competitive advantage in the marketplace. This is the source of inspiration that creates the demand and opportunity for knowledge work. Standardization that supports human communication and machine activation can formulate those insights into precise and complete definitions of the essential aspects of information exchange needed to make that work effective.

[9] This "80-20" rule is often heard in relation to software projects. I have heard others mention that it dates back to the ancient Greece, when somebody named Pareto articulated it as a principle. I do not have a reference for this, nor do I know in what context or what form it was articulated. I use it here in a colloquial sense based on anecdotal experiences. I am not aware of any statistical studies which have attempted to measure this.

4.4 Project Examples

In this section we will illustrate the use of templates as a basis for automation with examples from our project work. In the SoDA system (introduced in section 1.4.1), task templates were used to focus work on the constraints which a system design needed to satisfy. Checking constraints as changes were made to designs and suggesting actions which would fix violations was automated by the use of templates. We characterize this approach to automating design more formally as a process of delayed constraint maintenance. We discuss some of the limitations of this approach, including some flaws in the approach to “cut and paste” in SoDA which were corrected in the Axiant Prototype.

In the Axiant Prototype (introduced in section 1.4.2), task templates were used to focus work on the reuse of common constructs and patterns specific to database systems. A “cut and paste” metaphor was used as the basis for providing a language sensitive editor with automated layout and formatting, in which a comprehensive set of common code “fragments” could be easily selected and pasted to create procedural code for a Form, Report, or Utility. Standardized design patterns which captured common types of reports and forms were used as a basis for a Creation Assistant which could create Forms, or Reports in a matter of seconds. We discuss the insights into the nature of work for a specific domain which are required to achieve this level of automation

4.4.1 Task Templates for Design

The SoDA system was built to support and automate the task of software design. Design is very complex task and difficult to completely automate. There is no focused, precise, and well-defined characterization of the steps one should take in designing software. Some studies [Guindon 90] indicate that design is opportunistic and decentralized. Designers often consider several problems at once, fixing one or the other as a thought occurs to them. A small change in a design can have far reaching affects in many other parts of the design.

Our approach in this area was to avoid attempting to automate what was not well understood. Rather, we attempted to identify areas where automation could complement the work done by designers as they created and modified their design documents. Representing the components and relationships between components in a document model facilitated browsing of designs from different points of views. Transformation control within the document agent provided a flexible means for organizing and tracking actions as designers behaved opportunistically.

In order to represent the completeness and consistency of a design, a state profile was defined for design in terms of constraints. A system was consistent and complete if all constraints applied to the system were true. If it was inconsistent or incomplete, the particular constraint that was false would indicate the precise location and nature of the problem. A single change to a design could have many effects, both solving and creating several problems at once. It can be tedious, time-consuming and error prone to judge what exactly the effect of a design action was. Templates were created which automated the task of checking constraints to determine the effects of actions. When there is a design problem, there may be many ways in which the design can be fixed. Decision lists were created which provided an exhaustive list of all possible actions that could be applied to fix a constraint violation.

We will illustrate these two tasks with a simple example in which our Blackboard System design is modified to include a goal list in addition to an event list. An event list is a common mechanism for control in blackboard systems. The eventlist is part of the blackboard and contains events which the control unit can select to use as a focus when deciding which knowledge base to apply next. Knowledge bases when they are executed update the blackboard, and these updates are recorded as events in an eventlist. A goal list is a similar mechanism for control. The difference is knowledge bases must request the creation of a goal as an update to the blackboard. The control unit can select a knowledge base to respond to an event, or select a knowledge base to achieve some goal depending on whether an event or a goal is selected as the focus.

In Figure 19, a data flow view of our Blackboard System is shown before and after a design change. Initially the Cpanel of the Blackboard has a single part Eventlist. Eventlist has a module CreateEvent from which a data Event is given to the module CreateFocus inside the Knowledge Component. The BBAction of Bbupdate gives data Eventinfo used by module CreateEvent. A CopyModule action is applied to duplicate the Eventlist (and all its subparts) as a second part of the Cpanel. These duplicate modules have been renamed GoalList, CreateGoal and SelectGoal. The CopyModule command in SoDA, also copied all links (this is different then the algorithm previously defined for deep copy). All modules which call Eventlist or its parts will now call GoalList as well. The input and output for GoalList have been changed to Goal and GoalInfo.

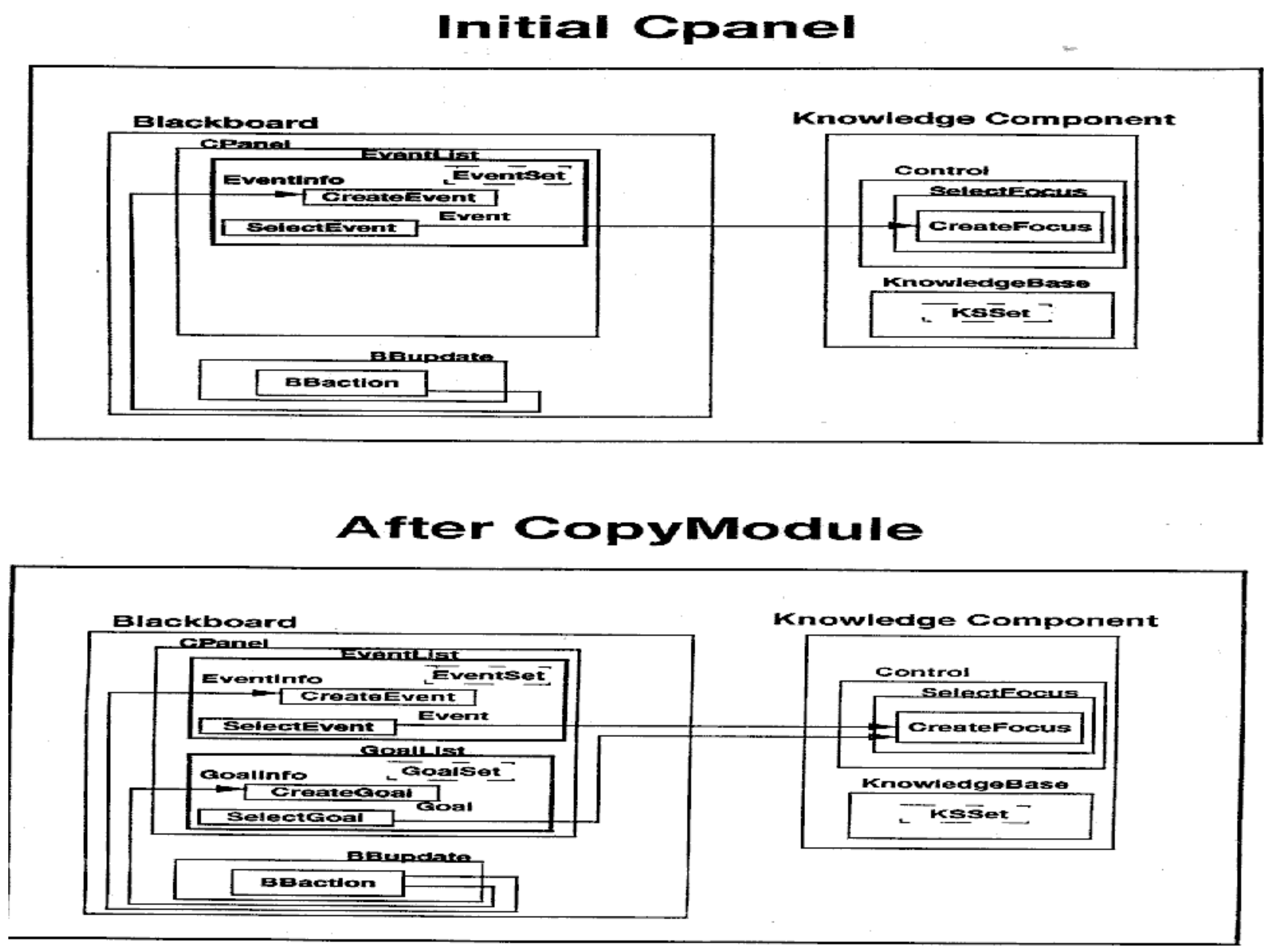
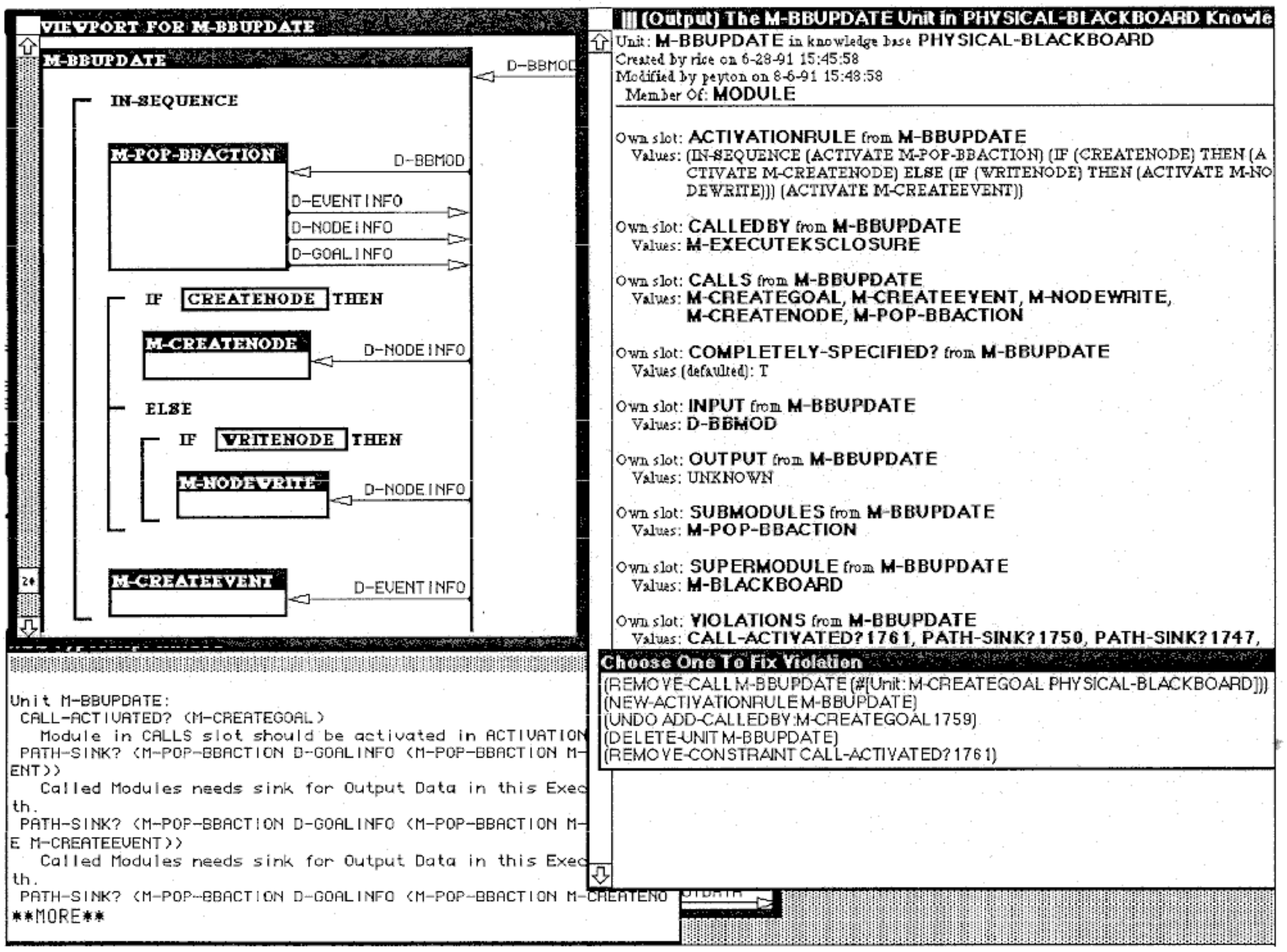


Figure 19. Design Change

Seen from the data flow view, we have achieved our objective. There are a number of actions still remaining, though, to complete the design change. Figure 20 is a screen snapshot which shows the control flow of Bbupdate. The call to SelectGoal has been added, but the ActivationRule for BBUpdate has not been changed. The CopyModule action assumes that we will want similar links when a module is copied, but the designer must decide exactly how that call will be incorporated into the ActivationRule. As a result, the top left pane which shows an action diagram view is based on the out of date ActivationRule. While the top right pane shows the description associated with BBUpdate indicates that is should have a call to CreateGoal.



(Output) The M-BBUPDATE Unit in PHYSICAL-BLACKBOARD Knowle

Unit: **M-BBUPDATE** in knowledge base **PHYSICAL-BLACKBOARD**
Created by rice on 6-28-91 15:45:58
Modified by peyton on 8-6-91 15:48:58
Member Of: **MODULE**

Own slot: **ACTIVATIONRULE** from **M-BBUPDATE**
Values: (IN-SEQUENCE (ACTIVATE M-POP-BBACTION) (IF (CREATENODE) THEN (A CTIVATE M-CREATENODE) ELSE (IF (WRITENODE) THEN (ACTIVATE M-NO DEWRITE))) (ACTIVATE M-CREATEEVENT))

Own slot: **CALLEDBY** from **M-BBUPDATE**
Values: **M-EXECUTEKSCLOSURE**

Own slot: **CALLS** from **M-BBUPDATE**
Values: **M-CREATEGOAL, M-CREATEEVENT, M-NODEWRITE, M-CREATENODE, M-POP-BBACTION**

Own slot: **COMPLETELY-SPECIFIED?** from **M-BBUPDATE**
Values (defaulted): T

Own slot: **INPUT** from **M-BBUPDATE**
Values: **D-BBMOD**

Own slot: **OUTPUT** from **M-BBUPDATE**
Values: UNKNOWN

Own slot: **SUBMODULES** from **M-BBUPDATE**
Values: **M-POP-BBACTION**

Own slot: **SUPERMODULE** from **M-BBUPDATE**
Values: **M-BLACKBOARD**

Own slot: **VIOLATIONS** from **M-BBUPDATE**
Values: **CALL-ACTIVATED?1761, PATH-SINK?1750, PATH-SINK?1747,**

Choose One To Fix Violation

```
(REMOVE-CALL M-BBUPDATE (#[Unit: M-CREATEGOAL PHYSICAL-BLACKBOARD]))
(NEW-ACTIVATIONRULE M-BBUPDATE)
(UNDO ADD-CALLEDBY M-CREATEGOAL 1759)
(DELETE-UNIT M-BBUPDATE)
(REMOVE-CONSTRAINT CALL-ACTIVATED?1761)
```

```
Unit M-BBUPDATE:
 CALL-ACTIVATED? (M-CREATEGOAL)
   Module in CALLS slot should be activated in ACTIVATION
 PATH-SINK? (M-POP-BBACTION D-GOALINFO (M-POP-BBACTION M-
ENT))
   Called Modules needs sink for Output Data in this Exec
th.
 PATH-SINK? (M-POP-BBACTION D-GOALINFO (M-POP-BBACTION M-
E M-CREATEEVENT))
   Called Modules needs sink for Output Data in this Exec
th.
 PATH-SINK? (M-POP-BBACTION D-GOALINFO (M-POP-BBACTION M-CREATENO
**MORE**
```

Figure 20. Catching and Fixing Design Violations

In the bottom left pane is a console which lists some of the effects of the CopyModule action. A number of inconsistencies have been created and flagged as constraint violations. Call-Activated? indicates that there is a discrepancy between the Activation Rule and the Call link to CreateGoal. The CopyModule action is not prevented from occurring, nor is it necessary for the designer to fix these constraint violations right away. The constraints are noticed automatically by SoDA and posted to the To Do list as violations still to be fixed. The console simply provides, as feedback, a list of constraint violations that have occurred, and a list of problems that have been fixed by the most recent action. Should the design wish to fix a constraint violation, it can be selected from the To Do list or from the description of the component where the violation has occurred. A list of suggestions will be displayed which indicate how the violation can be fixed.

Figure 21 shows how tasks in the document agent part of SoDA used templates to assist design in this manner. There was a task to check constraints and a task which could provide a list of suggestions for actions to resolve constraint violations. The task for checking constraints reduced the problem of checking constraints to a collection of Monitor Templates which defined a particular constraint to be checked when a particular event occured in a design document. Monitors were defined for all primitive events. Whenever a primitive transformation completed, the Transformation Control Task would send a notification for each primitive event that occurred. A decision list would match the event description to filter and select all the monitors which applied. For each monitor that applied, a Check action would be requested. The Transformation Control task would check the indicated constraint by calling a constraint action. If the constraint action returned false, then the Check action would update the To Do list with the constraint violation, and send a message to the console.

The task for Fixing Violations used a Decision List to create a list of suggestions to choose from. The task would receive a record of the constraint violation to describe the problem that needed to be solved. This was matched against a collection of possible templates. One template would be selected. This template created a list of transformations that could be replayed to fix the violation. This list would be presented to a designer. If a transformation was selected it would be replayed to create the desired effect.

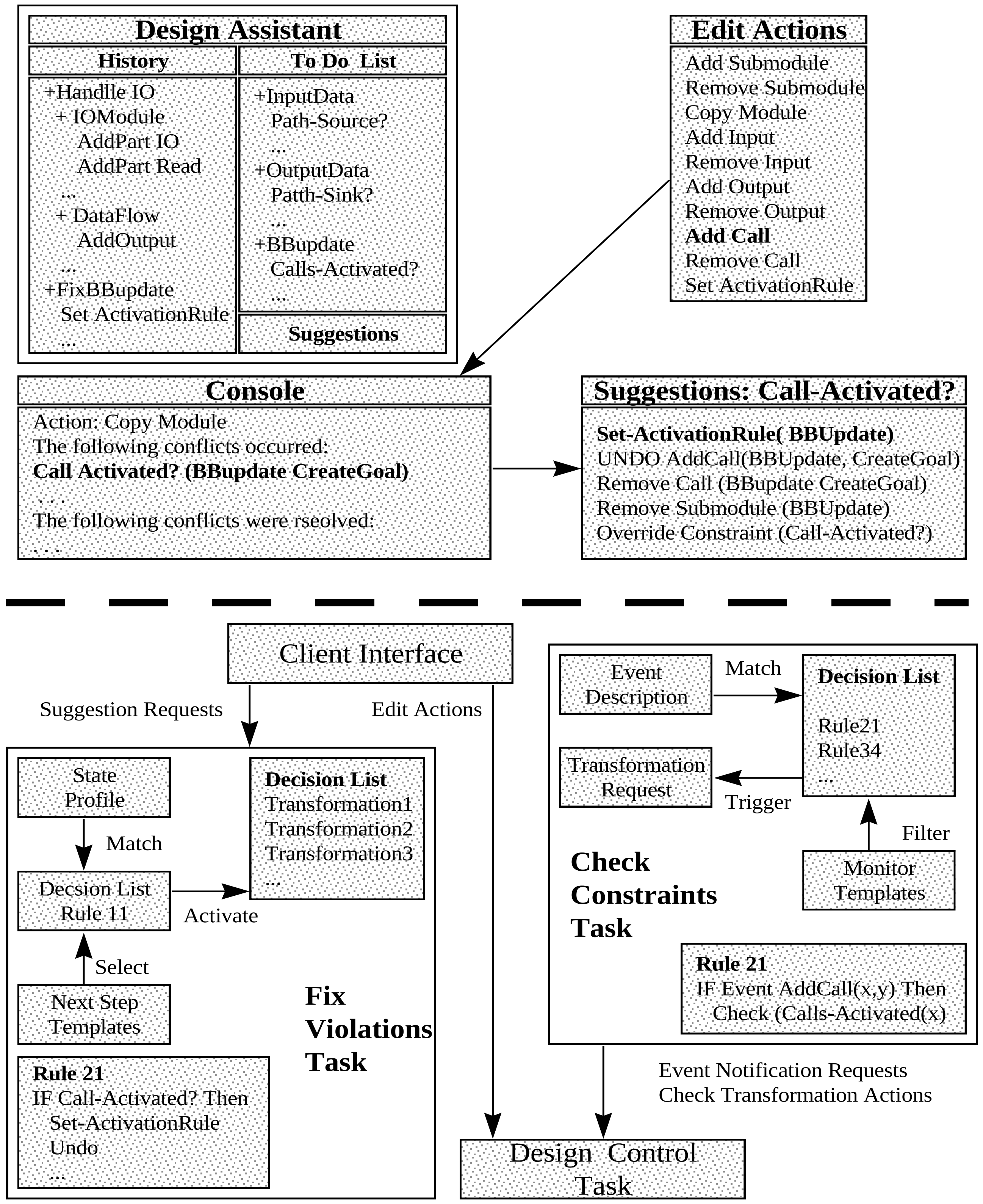


Figure 21. Interactive Design Tasks

This approach to automating design attempted to combine a powerful, but generic CopyModule action with flexible support for tracking and fixing constraint violations. The CopyModule action copied a specific module and all its specific dependency relationships, on the assumption that the new module would have similar dependencies. The important point being that it would have *similar* dependencies, not *identical* dependencies. As we have seen in our discussion of the implementation of semantic copy in the Axiant Prototype (section 2.4), special attention must be paid to the parts hierarchy of a component. In general, it does not make sense semantically to copy all dependencies. The definition of CopyModule almost ensures that there will be constraint violations when it is used. While this approach is flawed, copying and reusing existing design components in this manner does reflect common behavior amongst software designers. The use of templates to identify and fix the inevitable constraint violations greatly improved quality and productivity.

A greater degree of automation could be achieved, if there was a library of common components to choose from which were parametrized to indicate the relationships that needed to be established. It requires more effort and a good understanding of the specific domain to create such a library. In essence, such a library codifies the knowledge in that domain in a form that enables automation and raises the level of interaction.

4.4.2 Formalizing Delayed Constraint Maintenance

In SoDA, we have used constraints to define a valid goal state for the design document, but we have not insisted that the design remain in a complete or consistent state. Most formal approaches to software development have required that systems stay in a valid state throughout implementation. This is very restrictive and awkward. Transforming a design document while remaining in a consistent state is in fact impossible, if the requirements for the design are changing. In order to create a path which takes a design from an initial state in which all constraints are achieved, to a state in which different constraints are achieved, one will have to step through states in which not all constraints are achieved.

Our solution to this is not to limit the definition of validity to the state of the system being designed. During design it is only necessary that the state of a system be valid when the design process is complete. We will define validity in terms of the state of the design task, as well as the state of the system being designed. A system design is valid if all constraints will eventually hold for the completed design. We have a valid design task as long as we are aware of all constraint violations that are still left to be fixed. In any state of our design, either all applicable constraints hold or we have a list of constraint violations left to resolve with actions available that can resolve them.

In such a solution space, each step of the path is characterized by preconditions, postcondtions, obligations. Preconditions are constraints which hold and are true when a step is taken. Obligations are constraints which were violated when the step completed, post conditions are constraints which hold, or violations which were removed when the step completed. Each precondition must have a preceding step whose post condition created the constraint. Each obligation must have a succeeding step whose post condition fixed or removed the constraint violation.

e.g.:

Event CopyModule (Eventlist,Goallist)

> **PRE:** Part(Cpanel,Eventlist),Part(Eventlist,CreateEvent),Calls(BBupdate, CreateEvent) ...
> **POST:** Part(Cpanel,Goallist),Part(Goallist,CreateGoal),Calls(BBupdate, CreateGoal) ...
> **OBLIGATION:** Call-Activated? (BBUpdate, CreateGoal), Path-Sink? ....

Event SetActivationRule (BBUpdate, " ..... Call CreateGoal)

> **PRE:** Module(BBUpdate)
> **POST:** Call-Activated? (BBUpdate,CreateGoal) ...
> **OBLIGATION:** ...

4.4.3 Templates for Program Editing

In the Axiant Prototype, there was a desire to improve the facilities for editing source code associated with Forms, Reports, or Utilities. Initially, some thought was given to providing a structure editor. Previous experience with structure editors in the Muir Project (described in appendix A.7) had shown that while there was some good ideas in structure editing, they quickly became tedious to use. They were driven by the grammar constructs of the language not the cognitive constructs of the participant in the task. The approach on the Axiant prototype was to achieve the features of structure editing (automatic formatting, color coding, elision and navigation maps) but build them from templates that reflected the task of program editing. The metaphor for this was copy and paste. It is common among developers, when writing source code to find a previous example that can be copied and pasted as a close approximation from which to start.

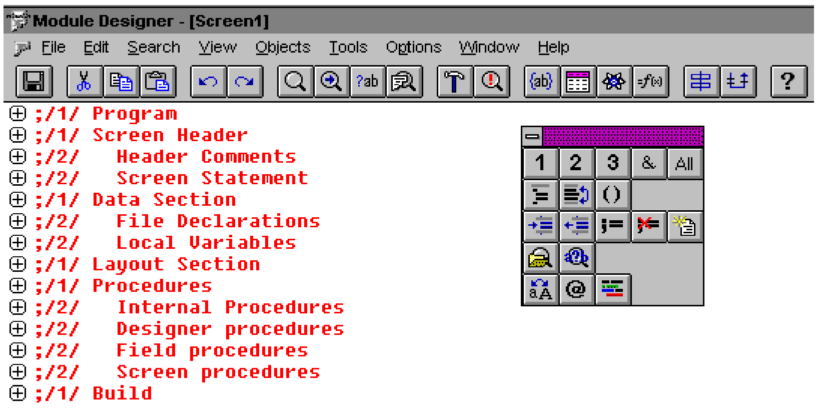
In many ways the declarative language used (PowerHouse) was well suited to this approach. There are some basic control mechanisms: IF, FOR , WHILE, but mainly a very rich collection of commands geared to specific aspects of database systems. Each command has a wide variety of options or subcommands. Most commands and options have parameters with a fixed set of choices. Although there is no language structure to support it, most PowerHouse developers organize programs into sections. The official courses for PowerHouse teach this organization. We supported this through the use of tagged or structured comments.

A library of templates was created. It was organized to list alternative command templates for an appropriate section. One command might have many templates, each showing a representative example of combining options. Once a command template was chosen, for each option there was a list of alternative templates that showed how different combinations of parameters could be specified. The middle snapshot of the Module Designer in Figure 22 illustrates the selection of a template to add an option to a Field command. Once an option was chosen, for each parameter there would either be a range list of values to choose from or a written description of what needed to be typed. An entire program could be quickly constructed simply by pointing and clicking and filling parameters. The user did not think syntactically. Rather, the template which was most representative of the code fragment needed was selected.

Pretty printing was based on the library of templates. Program text was pretty printed by matching the lines of text with existing templates. When a template was found that matched, the style of that template was copied to format and color the code which matched. As well, this process was incremental. Whenever something was typed, as soon as the user switched to a new line, the changed line, and any lines affected would be reformatted. This gave immediate feedback in terms of layout as to the validity of what was being typed. Syntax errors were obvious by formatting and coloring being askew. A user preference document was created to configure the manner in which pretty printing occured.

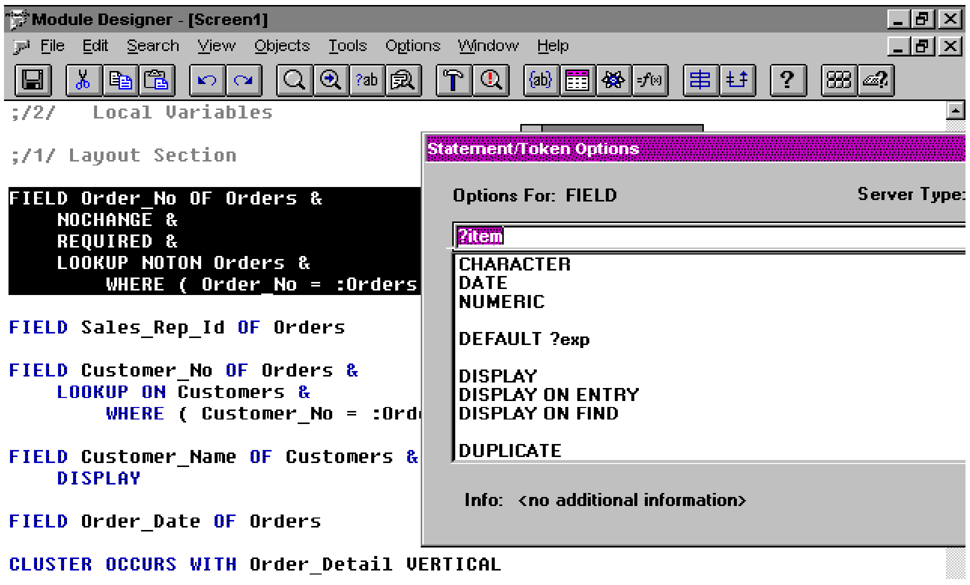
Structured comments were also used to facilitate navigation. Sections and subsections could be expanded and collapsed by double clicking on a structured comment. One could quickly browse an overview of source code with all sections collapsed. One could then drill down into the specific section of interest by double clicking to expand it. Figure 22 illustrates an overview, followed by drilling into the layout section.

In addition to structure comments, procedure maps were used that diagrammed the order of invocation for procedures associated with Fields, Forms and Reports. The last screen shot in Figure 22 shows the map of procedures for Forms. There are a large number of built-in predefined procedures provided by Axiant which define default behaviors when, for example, a screen was Initialized, or an Update action of a query was triggered. These can be customized and additional procedures can be added as Designer or Internal procedures. However the dependencies between procedures and the order of invocation can be quite complex. A map diagram defines this and provide a powerful means of navigation. The source code for a procedure could be viewed by clicking on the map, and then modified (certain critical procedures restricted this ability or warned of the dangers when a click occured). There was a natural mechanism within the map, to add custom procedures and show where they would be invoked with respect to other procedures.


Module Designer - [Screen1]
File Edit Search View Objects Tools Options Window Help
1 2 3 & All

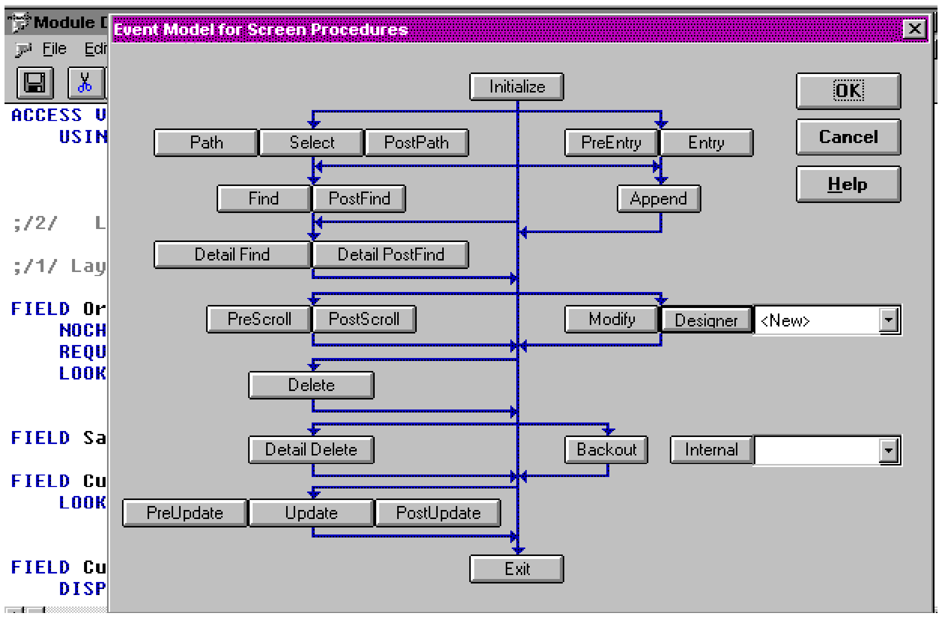
```
;/1/ Program
;/1/ Screen Header
;/2/   Header Comments
;/2/   Screen Statement
;/1/ Data Section
;/2/   File Declarations
;/2/   Local Variables
;/1/ Layout Section
;/1/ Procedures
;/2/   Internal Procedures
;/2/   Designer procedures
;/2/   Field procedures
;/2/   Screen procedures
;/1/ Build
```

Overview of Program Sections.



```
;/2/   Local Variables

;/1/ Layout Section

FIELD Order_No OF Orders &
    NOCHANGE &
    REQUIRED &
    LOOKUP NOTON Orders &
        WHERE ( Order_No = :Orders

FIELD Sales_Rep_Id OF Orders

FIELD Customer_No OF Orders &
    LOOKUP ON Customers &
        WHERE ( Customer_No = :Ord

FIELD Customer_Name OF Customers &
    DISPLAY

FIELD Order_Date OF Orders

CLUSTER OCCURS WITH Order_Detail VERTICAL
```

Selecting a Template to Add an Option



```
ACCESS U
    USIN

;/2/   L

;/1/ Lay

FIELD Or
    NOCH
    REQU
    LOOK

FIELD Sa

FIELD Cu
    LOOK

FIELD Cu
    DISP
```

Procedures Map for Navigation

Figure 22. Module Designer

4.4.4 Templates for Program Creation

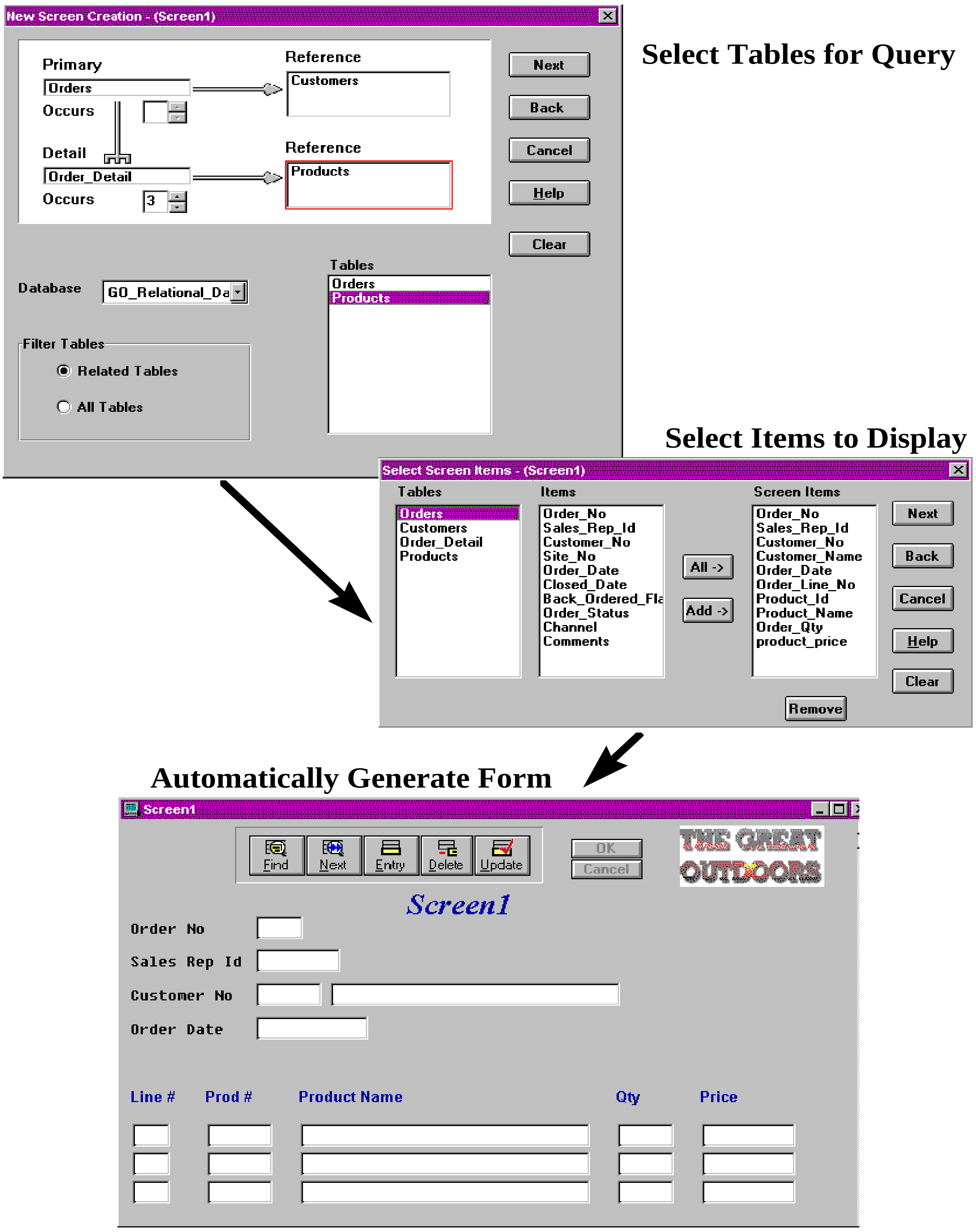
Command templates helped with the small steps involved in editing. There was also a requirement to automate the task of building an entire application. The idea was that many programs in an application follow a standard layout based on the underlying database model. The difficulty was in being able to formulate a characterization of those standard layouts that could be used to generate a program.

The first type of program that was tackled was a Form. The document model for Form was described in section 2.4.3. All Axiant forms are based on a database query or queries that may link several tables. This query always has a single main table. In theory, almost any combination of tables can be linked together in a query. However, most Axiant Forms that practitioners build follow a pattern in which there is one main table (say Order) which links to a few reference tables (like client) and a detail table (like Order Item). The detail table might also be linked to a few reference tables. Figure 23 shows the Creation Assistant user interface, which was based on this basic pattern.

The Creation Assistant instantiated this template by first presenting a dialog box in which the developer would fill in the appropriate tables. Decision lists were associated with each link. Any table could be chosen as a main table, but once that was selected, the possible detail tables and reference tables that could be linked were filtered and displayed for the user to select from. Once the tables for the query were selected, the Creation Assistant prompted the user to select which items of those tables should be fields in the form. Once that was done, the template was instantiated by a number of rules that described the placement of fields and the default commands, options and procedures to be used. In a matter of seconds, a complete form document would be generated, compiled and ready to execute enabling data entry for an application.

The process was configurable by the setting up of a master Form document that established the defaults for positioning titles, defining backgrounds, and even customizing default procedures. In Figure 23, a standard background with the logo of the company (The Great Outdoors) was automatically used and standard procedures for Find, Next, Entry, Update, and Delete were included. As well, there was a user preferences document to specify options related to building the application. These included options associated with generating a standalone pc system, a server-based system, or a client-server based system. Several different types of databases were supported and both UNIX and VAX servers were supported. Once the program was built, of course, it could be modified or refined using the various Designer interfaces (including the Module Designer and its command templates).

Creating a template of this nature requires a deep understanding of the domain acquired from actual experience working in the domain. It also requires insight or a fundamental conceptual breakthrough which recognizes common patterns in an abstract and high level manner so that complex tasks can be reduced to instantiating a few basic parameters. It also embodies an approximation that must be integrated into the domain of knowledge work flexibly. Not all tasks will fit the pattern. Even when they do, flexible and powerful mechanisms need to be provided so that the solution created is customized to a particular work environment.


Select Tables for Query

New Screen Creation - (Screen1)

Primary	Orders
Reference	Customers
Occurs	
Detail	Order_Detail
Reference	Products
Occurs	3
Database	GO_Relational_Da
Tables	Orders, Products

Filter Tables: [✓ Related Tables] All Tables

Select Items to Display

Select Screen Items - (Screen1)

Tables	Items	Screen Items
Orders	Order_No	Order_No
Customers	Sales_Rep_Id	Sales_Rep_Id
Order_Detail	Customer_No	Customer_No
Products	Site_No	Customer_Name
	Order_Date	Order_Date
	Closed_Date	Order_Line_No
	Back_Ordered_Fla	Product_Id
	Order_Status	Product_Name
	Channel	Order_Qty
	Comments	product_price

Automatically Generate Form

Screen1

Screen1

Order No

Sales Rep Id

Customer No

Order Date

Line #	Prod #	Product Name	Qty	Price

Figure 23. Creation Assistant

4.5 *Relationship To Other Work*

4.5.1 Research

Our understanding of tasks and task templates draws from early work on problem solving in artificial intelligence, especially with regards to the concept of a ‘solution space’, and transformations which take one from state to state [Amarel 68]. Automatic use of templates, though, is not required for effective and dramatic improvements in the quality and productivity of knowledge work. Design patterns, such as have been identified in object-oriented design [Gamma et al 95] illustrate this point. Knowledge-based approaches to automating software development use program transformation rules [Goldberg 86] to automate the implementation of formal specifications. Our definition of delayed constraint maintenance are based on the concept of obligations introduced in [Perry 87]. In the KIDS system [Smith 89] general templates for techniques like Divide and Conquer were applied to formal specifications to automatically generate implementations for sorting algorithms like quicksort. The theory and mathematical logic used to create those templates is described in [Smith & Lowry 89]. The Programmer’s Apprentice project [Waters 86] used templates for software algorithms called cliches.

4.5.2 Industrial Practice

The concept of template is becoming more common in systems or products that are built in industry. It is quite common for word processor, spreadsheet, and presentation programs to be packaged with a wide selection of predefined document templates for common applications. This thesis, for example, was created using a “thesis” template packaged with Word 6.0 that predefined the formatting for sections, headers, footnotes, figures, table of contents etc. All the major office suites (Microsoft Office, Lotus SmartSuite, WordPerfect Office etc.) have facilities for recording a sequence of transformations as a macro which can be replayed anytime a similar solution is required.

Software producers are also using templates to create more sophisticated support for tasks commonly performed with their products. Spelling checkers have been common for a number of years, but recently they have become more sophisticated. An option can be turned on, so that spelling is monitored while one is typing, automatically correcting known errors as they occur. Relatively complex tasks like creating a chart with a spreadsheet program, setting up a schedule in a project management tool, or creating a data-entry form in a database tool are supported by some form of task assistant. In the simplest case, on-line help can provide a scripted description of how the task should be performed, stepping a user through the actions that should be requested. In other cases, automated support is provided in a manner similar to the description in the Axiant project by a sequence of dialogs that defines the task to be performed in a structured manner using decision lists which the user selects from to formulate their particular solution.

There are many case tools on the market which support system design (usually from an object-oriented or data modeling perspective). Most use constraints to ensure a consistent and complete design as SoDA did. Usually, they are handled in one of two ways. Either a separate process can check the whole design all at once cataloging a list of errors (usually just before or as a part of code generation), or the system is proactive preventing the user from introducing an error. To the best of our knowledge, none has the flexibility to flag and track errors as they occur or provide suggestions as to how the problems might be addressed.

There are many products on the market which support database system development. Language sensitive editors which provide color coding and automatic formatting are common, but none have a comprehensive set of alternative code fragments available directly in the editor, as Axiant does. Similarly, creation assistants which provide a sequence of dialog boxes to guide a developer in the creation of a database program are common. None, however, supports configuration of common functionality like logos for forms, pre-defined functions, standalone vs. client-server architecture etc. in the manner that Axiant does.

5.0 Conclusion

Our thesis has been that knowledge work can be automated by the embodiment of its documents and tasks in software. To illustrate that principle, we have presented an architecture which implements document models and task templates within a document agent as a basis for software automation of knowledge work. This has proven to be quite successful in our project work for automating the individual tasks that a knowledge worker might perform. These individual tasks, however, are usually performed as part of a larger process that serves the objectives of a human institution. In this chapter, we will conclude our thesis with a discussion of how our work should be understood within this broader context. This will include an analysis of a special type of document agent called a task coordinator which can be used to manage a knowledge work process and coordinate the tasks performed by knowledge workers.

5.1 *Electronic Work*

"'What did the mirror say?

... It's done with people.'"

Ken Kesey, in Tom Wolfe, The electric kool-aid acid test, Bantam Books, New York, 1969

5.1.1 Vision

We have presented an architecture for the automation of work that conveys knowledge, but we have not discussed to whom the knowledge is conveyed or for what purpose. It is a premise of this thesis that knowledge work takes place within the larger context of our human institutions. Knowledge work is performed to convey knowledge to coworkers and clients in a process that achieves the objectives of those institutions. When work is performed in an electronic medium, those objectives can be achieved by the coordination of information exchange solutions created by participants in one or more electronic domains. Electronic work, which we define as *the embodiment of a knowledge work process in software*, enables the objectives of a human institution to be achieved by conveying knowledge through its *coordination of information exchange solutions*.

It is beyond the scope of this thesis to examine software automation in the context of the objectives which are pursued by human institutions. It is within the scope of this thesis to address the manner in which tasks and documents automated by software can be coordinated in an electronic medium. What does software need to provide in order to automate the coordination of solutions within a process of knowledge work? What benefits can be gained and what concerns need to be addressed?

A knowledge work process is an established pattern of interactions between participants in one or more domains of knowledge work. A task performed by one participant creates information that conveys knowledge to another participant that is used in the task they are performing. The performance of those tasks is *assigned* to participants within a human institution in order to achieve certain objectives. Tasks are *managed* so that information from one task can be transferred to and used by another task in a timely fashion. The results of tasks are also *controlled* to ensure that the objectives of the institution are being met.

A knowledge work process can be automated by software that creates a task coordinator. A task coordinator is a special type of document agent. It executes tasks and document models on behalf of a facilitator who is responsible for the objectives that a knowledge work process is intended to achieve. A task coordinator implements tasks whose actions can be used to assign and coordinate tasks performed by other document agents. The document models of a task coordinator define schedules and reports to capture the results received from tasks.

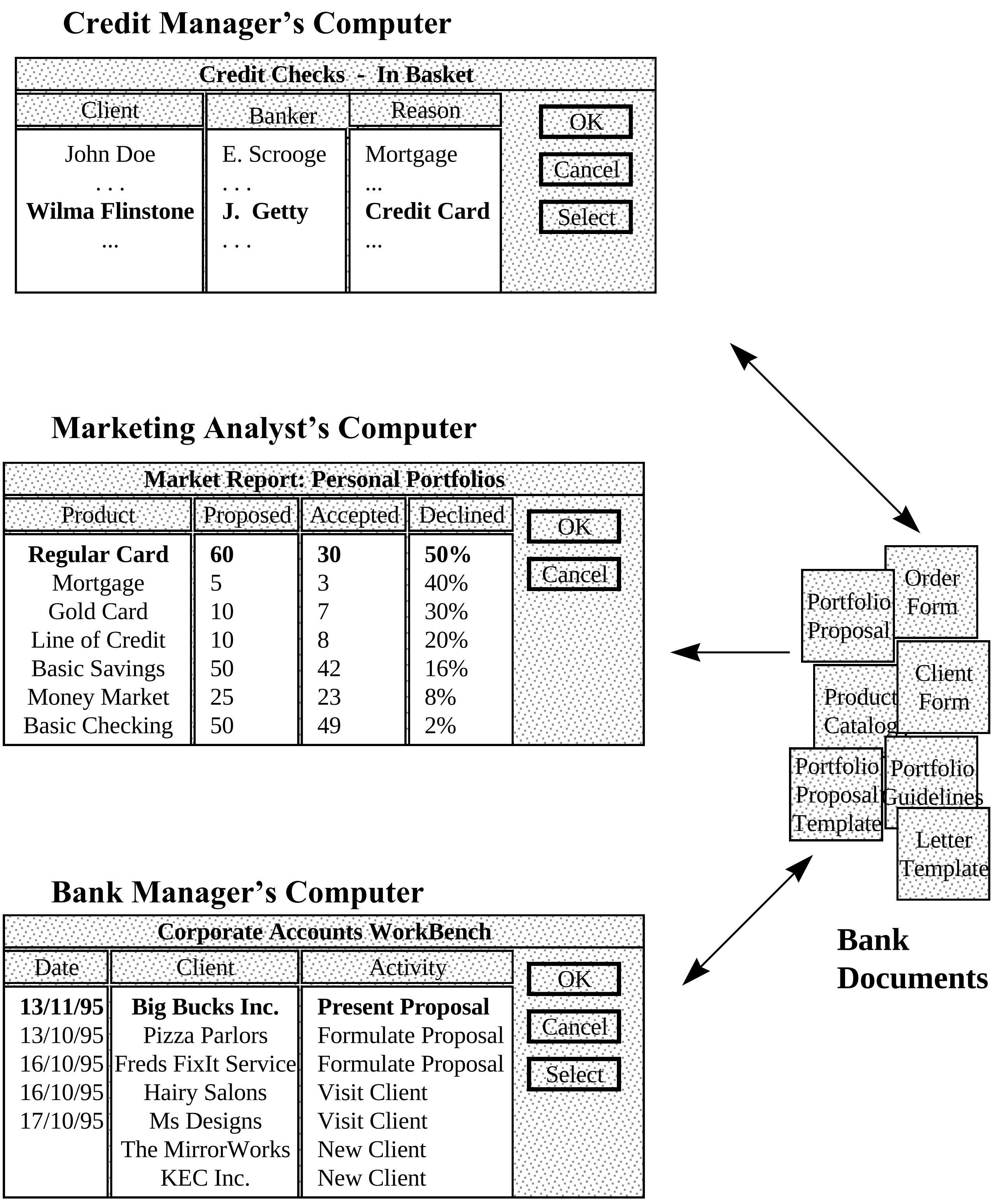
5.1.2 Scenario

We will illustrate the requirements for a task coordinator, by considering our bank scenario. After Wilma agrees to the Portfolio Proposal that the banker has created, there are a number of tasks which have to be performed, so that the orders associated with the Portfolio are filled. The banker, of course, will fill in the appropriate order forms, but other participants will be involved. Among other things, two credit cards are being ordered for Wilma. Normally, whenever credit cards are ordered, a credit check is performed. We can see in Figure 24, that there is an In-Basket Document on the credit manager's computer which lists all the credit checks that have not yet been performed. It lists the name of the client, the banker who is responsible for the order, and the reason for the credit check. We can see that a credit check is needed for Wilma because credit cards are being ordered. When she is ready, the credit manager will select one of the entries in the In-Basket and perform the task of doing a credit check for that entry.

Meanwhile in another part of the bank, a marketing analyst is reviewing the results from the recently created Personal Portfolios program. The bank has a wide range of products, but it has been difficult for both employees and clients to understand what selection of products would best suit a client's needs. The bank put together the Portfolio Guidelines document to assist in the task of product selection. Some of the results shown so far are displayed in a Marketing Report document on the marketing analyst's computer. For each product, we can see how many times it was proposed to the client, how many of each product were accepted, and what percentage of proposals were declined by the client. We can see the problem with credit cards that was mentioned earlier. Half the clients rejected the initial proposal of a single credit card. In fact, more credit cards were accepted than proposed. The single credit card was rejected, because married couples wanted two cards, not one. The Market Report organizes the results of many tasks performed by many bankers over a period of time.

Finally, in another part of the bank, the bank manager is talking to our banker. It appears that one of the senior bankers at the bank has just given notice that she will be leaving to pursue a career as a junk bond trader on Wall Street. She has been responsible for several key corporate accounts at the bank. The bank manager has decided that this might be a good time to move our banker from personal accounts into a position with more responsibility involving corporate accounts. On the bank manager's computer, they are looking at the WorkBench document for the banker who is leaving. Listed in the workbench is the tasks that the banker had been working on. Today, there is a proposal to be made to Big Bucks Inc. The banker is explaining that the Personal Portfolios program was actually modeled after the Corporate Portfolios program, so many of the tasks that the banker will be responsible for are similar. There is a product catalog, guidelines document, and letter templates that can be used in approaching customer accounts. There are differences, though. The bank manager will perform the presentation today, while the banker watches and is introduced to the client.

If we view this scenario as the coordination of information exchange solutions, then some observations can be made. The coordination of solutions requires an *assignment environment*. The task of a credit check for Wilma is assigned when the banker's proposal is accepted and an order for credit cards is placed. A task coordinator assigns tasks to different participants by triggering information exchanges in documents like the Credit Checks In Basket, and the Corporate Accounts Workbench. A task coordinator also requires *solutions management*. Many solutions may be performed at the same time within certain time frames. The individual steps of a solution may need to be tracked as well as the final outcomes. The Corporate Accounts Workbench shows several clients for whom a Portfolio Proposal will be made and when the next step in that process is scheduled. Finally, a task coordinator requires *results control*. The Marketing Report for the Personal Portfolios Program tracks results at each step where a product was recommended. The Letter Template and Portfolio Guidelines document are also a form of results control in that they are designed to assist bankers in creating results in a way that will achieve the bank's objectives.



Credit Checks - In Basket

Client	Banker	Reason
John Doe	E. Scrooge	Mortgage
. . .	. . .	...
Wilma Flinstone	**J. Getty**	**Credit Card**
...	. . .	...

Market Report: Personal Portfolios

Product	Proposed	Accepted	Declined
Regular Card	**60**	**30**	**50%**
Mortgage	5	3	40%
Gold Card	10	7	30%
Line of Credit	10	8	20%
Basic Savings	50	42	16%
Money Market	25	23	8%
Basic Checking	50	49	2%

Corporate Accounts WorkBench

Date	Client	Activity
13/11/95	**Big Bucks Inc.**	**Present Proposal**
13/10/95	Pizza Parlors	Formulate Proposal
16/10/95	Freds FixIt Service	Formulate Proposal
16/10/95	Hairy Salons	Visit Client
17/10/95	Ms Designs	Visit Client
	The MirrorWorks	New Client
	KEC Inc.	New Client

Figure 24. Coordinating Tasks at Banks 'R Us

5.2 The Coordination of Information Exchange Solutions

In this section we will specify the functionality that a task coordinator needs to provide in order to support the automation of a knowledge work process. Our intention is to consider how task coordination can be used to manage the objectives that a human organization is trying to achieve. In this regard, we have been influenced by issues and requirements which arose in our project work. Our experiences with organizations who have been adopting software automation technology (Appendix A.2, A.3, A.4) has been particularly instructive.

We will treat a task coordinator as a special kind of document agent whose behavior is configured by document models and task templates. There is, however, a shifting of levels in considering the domain of information exchange created by a task coordinator. A task coordinator does not automate a domain of knowledge work. Rather, it automates a domain of management of knowledge work. The documents of a task coordinator are schedules and reports which are used to coordinate and track the information exchange solutions which knowledge workers create as they perform tasks. The document models for these documents will need to model the process organization in order to determine who should perform what task when.

The participant who interacts with a task coordinator is a special kind of participant which we have chosen to call a facilitator. The facilitator is responsible for ensuring that the objectives of a human institution are being met by the results of tasks performed in a work process. Feedback in the form of results can guide how schedules and tasks are configured by the facilitator. In that regard, the task coordinator *informates* at the same time as it *automates* a work process [Zuboff 88].

5.2.1 Assignment Environment

> A task coordinator is a document agent whose execution provides an assignment environment in which the creation of information exchange solutions is coordinated between participants associated with one or more domains of information exchange.

In a knowledge work process, there are solutions which must be created in order to achieve objectives and there are participants which are capable of creating those solutions. A task coordinator provides an environment which enables the assignment of information exchange solutions to participants associated with one or more document agents. That assignment is performed on behalf of participants we will call *facilitators*. A facilitator is a participant responsible for ensuring that a knowledge work process achieves its objectives.

Process Organization

A process organization maps objectives to a combination of information exchange solutions which are designed to achieve those objectives. It also maps information exchange solutions to participants and their associated document agent(s) which are capable of creating those solutions. The task coordinator executes document models and tasks that implements its coordination of solutions and participants in terms of this process organization. Task templates can be used to create a process organization. A process organization defines each of the steps in a process as an information exchange solution that will be performed by some participant. State profiles, transformation rules, and decision lists can be used to configure which solution is performed by which participant at each step. A process organization differs from a task template in that the final states it defines for a process can not accurately be called a “goal state” for the process. Objectives are often stated as a measurement of results which accumulate over a large number of applications of the process organization. Some of those results may also be measured outside of any domain of information exchange. (e.g. Measuring how many clients say that they were satisfied with the service provided.)

Communication

Communication establishes an interaction with participants through their associated document agents. The task coordinator executes interfaces that implements the exchange of information required for coordinating information exchange solutions. Participants receive documents which define the solutions to be created and the results which are expected. The task coordinator receives the results of each solution as well as to the results of individual steps within a solution which may be relevant to the measurement of objectives.

Example

In our scenario, the bank has an objective of ensuring clients and employees understand how the bank's products can meet their needs (presumably because then the bank will receive more business). A process has been designed around tasks like Enter Client Information, Formulate a Portfolio Proposal, Enter Orders, and Perform Credit Check. The bank has implemented that process as electronic work which automates the sequence of tasks and collection of results. Different tasks in the process have been assigned to different participants. The credit manager will perform credit checks, while there may be several bankers performing the tasks associated with providing service to a client.   The tasks assigned to a participant are viewable in the Credit Checks - In Basket and the Corporate Accounts Workbench. Both documents are based on a decision list from which the credit manager or the banker selects the next task to be performed. There are many applications of the process in progress at any one time. The banker can choose to work with one of several clients each of which is at some step in the process. The order of steps in the process is important as well. Presumably the task coordinator which controls these lists will not allow the tasks Enter an Order or Formulate a Portfolio Proposal to be performed until the task Enter Client Information has occurred.

5.2.2 Solutions Management

> A task coordinator is a document agent whose documents are used to manage the state of information exchange solutions created by participants associated with different domains of information exchange.

In a knowledge work process, the combination of solutions created to achieve objectives is dependent on the timing and results of each of the solutions created. A task coordinator manages the timing and results of solutions assigned to participants associated with one or more document agents. That management is performed on behalf of a facilitator for the knowledge work process.

Schedules

A schedule determines the combination and sequencing of solutions created for a particular application of a process organization. The task coordinator provides actions to create access schedules implemented as documents. A schedule can contain decision lists, which are used to order and filter the next solutions to be created in the process. Those steps are assigned to different participants. The schedule defines the initial state for the assigned solution and the final state which is reported as a result. Time constraints or defaults can be associated with both the start and the finish of a solution. A schedule may define a final result for a solution that does not finish within its time constraints. It may also initiate another step in the process to address the problem. A monitor can be used to schedule information solutions when certain reports occur.

Reports

Reports are used to record the results in the application of a process organization. They are also used to collect and quantify results over many applications of the process organization in order to measure the achievement of objectives. A report can record results from a document agent at each step in a solution as well as the final state. The task coordinator provides actions to access reports defined as documents. The determination of which results to record is made by a facilitator for the work process. Participants or a facilitator may perform solutions not prescribed in a schedule as they deem necessary in response to the information contained in reports, especially if the reports indicate that objectives are not being met.

Example

Returning to our scenario, we can see that the In-Basket and Workbench documents show schedules that organize the bank's knowledge work process. The dates in the Workbench indicate when solutions are expected to be completed. The Workbench may monitor time lapses. For example, if a proposal is presented to a client and they do not respond within a certain amount of time, then the schedule may dictate that the result of presenting the proposal was that the client declined the proposal. The results reported from each Workbench solution influence the solutions assigned to the credit manager. Not every client participating in the process has a credit check performed. Only if a loan is involved will a credit check be performed. The entries in the In-Basket do not have dates associated with them, but it might be appropriate to report the time taken in order to ensure reasonable speed in enacting orders.

The Marketing Report document collects and displays reports from the bank's work process. There are individual steps within the task of formulating a Portfolio Proposal that are relevant to the objectives of the process. Each recommendation of a product was reported along with the client's response. One objective at the bank is to ensure that clients and employees understand how to match products to needs. When a recommendation is declined this might indicate that clients do not understand the product, or that the bank did not have a good product for certain needs. Once the analyst realizes that the recommendation for a single card is not appropriate for married couples, actions can be taken to adjust the process. A memo can be written to bankers instructing them how to override the recommendation in the Portfolio Guidelines. Eventually, the Portfolio Guidelines document can be updated. As we saw in the scenario in Chapter 4, the analyst can also propose a marketing campaign to create even more awareness of the bank's products.

5.2.3 Results Control

A task coordinator is a document agent whose documents configure information exchange solutions to control results created by participants associated with different domains of information exchange.

Results can be controlled in a knowledge work process to the degree that the solutions created in the knowledge work process can be adapted or evolved. Solutions are determined by the process organization used to schedule and report their occurrence, and they are determined by the types of documents and tasks which are used to create them. A task coordinator controls the results of solutions that are assigned to participants through its transformation of documents which configure process organization, tasks, and document models. That control is performed on behalf of a facilitator for the knowledge work process.

Objectives

A process organization defines a plan for achieving objectives. The configuration of a process organization determines what schedules are implemented to create results and it determines what reports are implemented to measure objectives. The task coordinator provides actions to transform the scheduling and reporting of information exchange solutions. A plan to achieve objectives can be improved by changes to the timing and combination of solutions performed. Objectives can be measured more accurately by reporting different or more detailed results. As a knowledge work process adapts or evolves, objectives may change. The definition of objectives for participants in terms of results expected from individual tasks can affect the manner in which a knowledge work process unfolds.

Distribution

Document models and tasks at each of the document agents in the knowledge work process define a solution space in which the results used to measure objectives are created. Task templates define reusable solutions in this space. The task coordinator provides actions to distribute and synchronize documents which configure the document models, tasks, and task templates used by document agents to create information exchange solutions. The manner in which solutions can be configured is dependent on the implementation of the task coordinator and document agents involved. Ultimately, solutions can be configured by a reimplementation of each document agent and its parts. In these cases, a task coordinator can control the transfer and versioning of the software executables which implement document agents.

Example

The Marketing Report document in our scenario was used to determine that objectives were not being met - at least with respect to credit cards. In general, it is a complex task to identify and quantify objectives in a manner that can be captured in results reported. What percentage of declines would indicate that objectives are being met for mortgages? Another report might have been used to determine that there was client dissatisfaction due to delays in receiving credit cards. For clients with an established relationship at the bank a credit check may be unnecessary. The scheduling of credit checks for credit card orders could be reconfigured so that it is not triggered when the Client has a mortgage with the bank.

When the task templates used to create solutions are reconfigured, then the situation is a little more complex. When the bank updates the Portfolio Guidelines document to include the new rule concerning credit cards, it must ensure that the document is distributed effectively. Each banker must receive an updated copy. A decision must also be made, about how to handle existing solutions. Should the Formulate Proposal task be performed again with the new rule? Should clients who have already accepted a Proposal be contacted and informed about the new marketing campaign where two credit cards are provided with only an annual fee being charged for one? The version of the Portfolio Guidelines in use must be coordinated with each banker and with each solution created. Presumably the next iteration of reporting will be coordinated to measure the impact of the marketing campaign and the change to the Portfolio Guidelines document.

5.3 Architecture

The architecture we have outlined for a task coordinator is portrayed in figure 25. The task coordinator assigns solutions, manages solutions, and controls results through its interactions with a facilitator and other document agents. Documents agents within the knowledge work process must cooperate with the task coordinator. The process organization which a task coordinator applies is defined in document models and task templates. The state of the knowledge work process that the task coordinator is managing is captured in its documents. There are documents which describe the schedules of work in progress and documents which report the results. There are also documents which describe the configuration of the process organization and its relationship to the objectives it is intended to achieve. Finally, there are documents for distribution to other agents that configure the document models, tasks, and task templates used to create solutions.

This architecture has not been implemented in the experiments that were performed for this thesis. As such, it should be considered as a proposal which still needs to be proved. This proposal does, however, originate with problems and issues that arose in the context of our experiments. Automation of the tasks performed in knowledge work requires standardization in the definition of problems and the creation of solutions. The determination of what to standardize and how is only effective when there is an understanding of how the solutions created by tasks are combined in a process organization to achieve objectives. Further, the benefits that are achieved by automating tasks are restricted in their practical application, if the manner in which solutions are communicated and coordinated within the overall work process are not automated. Finally, the introduction of such automation into an organization can be expensive and time consuming. The ability to show a measurable impact on the objectives of an institution can justify the endeavor.

The task coordinator is a natural evolution in the overall architecture which we have been developing in which to express an understanding of how software can automate knowledge work. When the orientation of this thesis shifted from a focus on better software tools to a focus on knowledge work, it became inevitable that the issue of coordinating solutions would have to be addressed.

Figure 26 diagrams three stages in that evolution. In the first stage, there was the realization that we were attempting to build tools that assisted participants who worked with documents. Their interaction with documents could be automated by the creation of a document model which defined that interaction as a system of information exchange. Once our experiments approached any sort of complexity in the types of interactions which were taking place or the quantity of documents and participants, it became obvious that a document agent was needed to manage the state of knowledge work captured in documents and the transformations of that state which were occurring.

In the second stage of evolution for our architecture, there was the realization that the work we were attempting to automate was problem solving that could be decomposed into commonly occurring subproblems for which there were known solutions. The creation of solutions in knowledge work could be automated by task templates that parametrized existing information exchange solutions for reuse in tasks. Once our experiments approached any sort of complexity in the types of problems which were being solved or the quantity of tasks and participants, it became obvious that a task coordinator was needed to coordinate the state of the overall work process captured in the solutions created and to control the results that were being achieved with those solutions.

It would seem that a third stage of evolution might be possible for our architecture, but that would take us beyond the scope of this thesis. If the patterns which enabled automation in the first two stages were document models and task templates, then it would seem that such a pattern in the third stage would be something like process organization. To the degree that an institution can be viewed as a knowledge work process, benefits can undoubtedly be gained from automation based on a standardization of process organization. That standardization could only be adequately addressed, if we understand software automation in the context of the objectives which are pursued by human organizations.

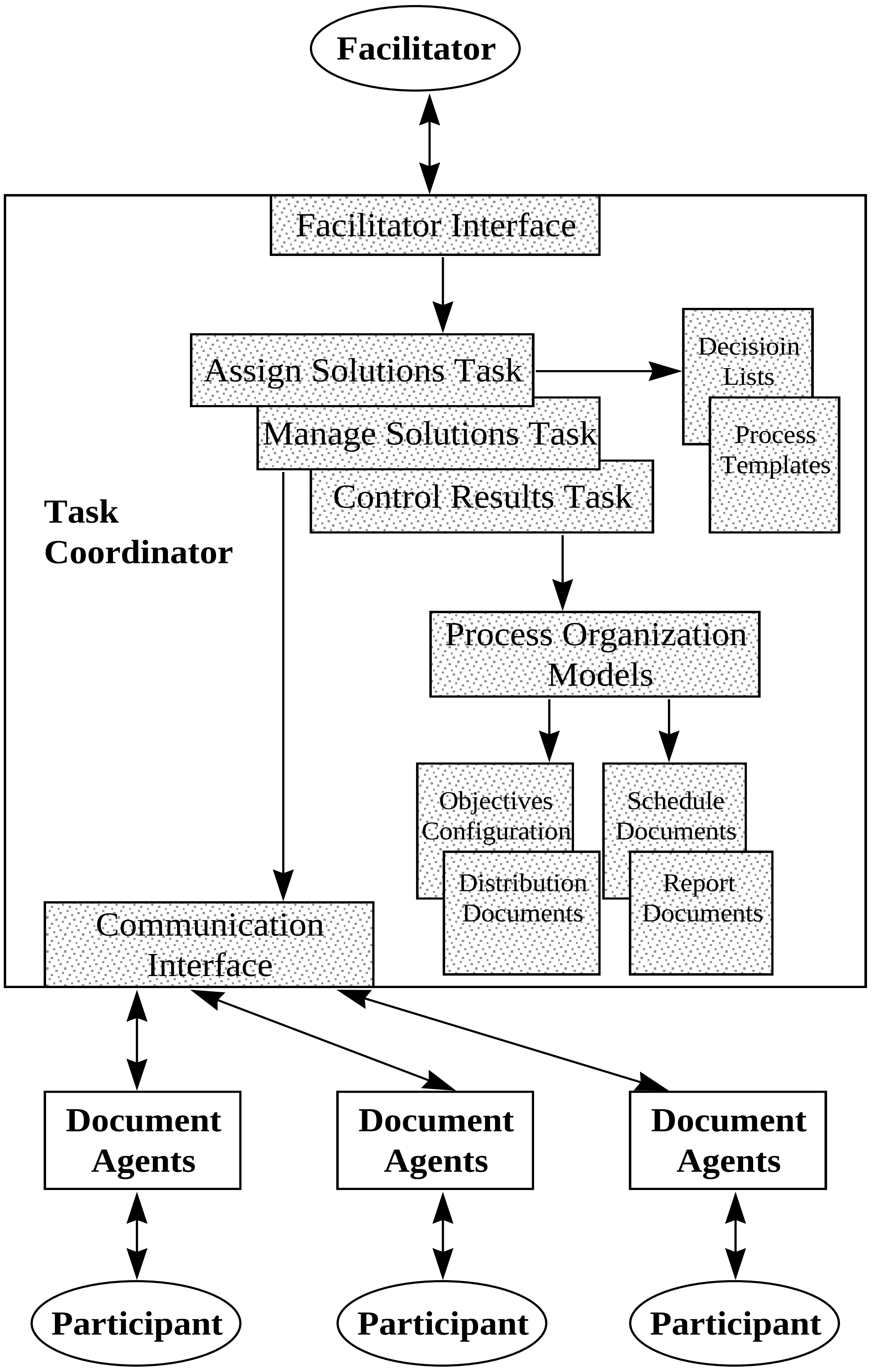


Figure 25. Task Coordinator Architecture

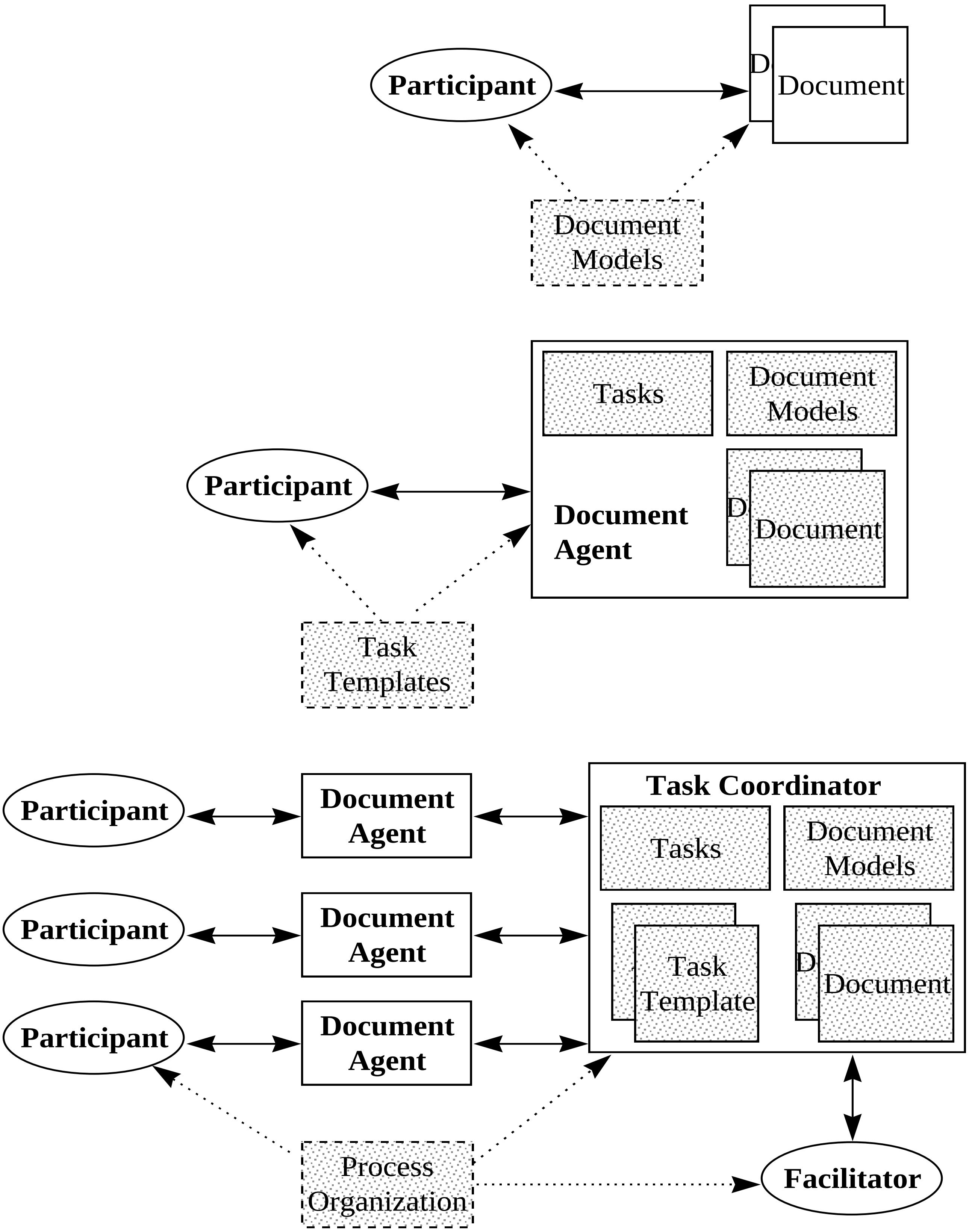


Figure 26. Evolution of Architecture

5.4 Lessons from Project Examples

Our thesis has been that knowledge work can be automated by the embodiment of its documents and tasks in software. The critical insight that has been reinforced by our project work is that documents are the fundamental basis for software automation of knowledge work. Documents, whether they describe banking information, software designs, or database systems, define the conceptual organization of knowledge work. Software organized around document models in the form of a document agent reflects that conceptualization and provides an electronic domain where knowledge work can be automated. We have also seen in our projects that document models in a document agent architecture are a sound basis for generating configurable and portable software systems.

The other critical insight that our project work has demonstrated is that the tasks knowledge workers perform should be understood as the creation of information exchange solutions. Tasks, whether they are performed by bankers, designers, or developers can be defined in terms of document transformations. Software organized around task templates reflects that understanding of knowledge work and provides a mechanism for characterizing standardized solutions which can be reused in the performance of knowledge work. An order of magnitude improvement in both productivity and quality can be realized when a template that captures a fundamental pattern of work is parameterized for configurability as was demonstrated in the Creation Assistant in the Axiant Prototype.

The primary lesson learned from our project work is that automation of knowledge work requires an evolution in the organizational processes which support and manage the performance of tasks. Work must be standardized in order that document models and templates can be defined. The overall knowledge work process will not benefit when individual tasks are automated unless there is an effective mechanism for assigning tasks, coordinating the solutions created, and measuring results. In the Axiant project, there was increasing market demand for integration with project management tools as well as configuration management tools in order to manage and track work performed by a team of developers. It has been our experience that organizations across almost every domain of knowledge work are adapting their work processes in order to leverage software automation.

In the short term, our project work would indicate that there are a number of areas where work should be undertaken to improve and extend the ideas presented in this thesis. More direct support for creating document models and task templates, either in the form of language mechanisms or tools needs to be provided. These should provide flexible support for adding new components and configuring existing ones. This needs to be understood within the context of creating a framework for an application specific domain. SoDA, for example, could have provided a greater degree of automation for designing blackboard systems if a library of standard blackboard system components and configurations had been defined.

Creating such a library, in effect, defines and standardizes a domain of knowledge work. It should be accessible to the workers and domain experts who are responsible for the evolution of that domain and be integrated into the overall work process. Better mechanisms for integrating document model and task template definition with on-line help facilities which support workers in understanding and participating in a work process would be beneficial. Work must also be undertaken to build a task coordinator and understand better how document models and task templates can be defined to improve the management and coordination of entire knowledge work processes.

In the long term, however, improved software automation will be dependent on developing better theories of management for business processes. The nature of work has been changed fundamentally by the shift from paper-based processes to electronic-based processes. Our understanding of business and business organization needs to adapt to reflect this change and unleash its potential. Software automation is a tool to be applied once that understanding takes form.

5.5 Final Remarks

The architecture for the software automation of knowledge work which we have presented in this thesis serves two purposes. First, we have defined a conceptual framework in which the technology of software automation can be understood. That framework provides a scientific foundation which can be used to guide and inform the practitioner in using and further developing the technology of software automation. Second, we have characterized the nature of work that is automated by software. That characterization provides a perspective on work which can be used to guide and inform human institutions as they adapt to new software automation technology. Our understanding of both software technology and the nature of work has been transformed in the process. We will conclude our thesis by discussing this understanding in terms of a knowledge work automation spectrum and by addressing some benefits and concerns associated with the spectrum.

5.5.1 Knowledge Work Automation Spectrum

Our goal has been to create software that supports an information level interaction that appears knowledgeable to participants as they perform work. Most software used in knowledge work today is restricted to a data level interaction that appears informative to participants as they perform work. In figure 27, we revisit and extend our diagram from Chapter One which situated the work in this thesis on a spectrum of automation. Increasing degrees of automation changes the level at which participants can perform work. Documents organize data in a manner that participants can interpret as information that can be communicated. A document agent groups actions into tasks in a manner that enables participants to work with information that reflects the state of their domain of work. A task template organizes information in a manner that participants can interpret as knowledge that can be used to solve problems.

It is reasonable to consider the interpretation that should be given at higher and lower levels of the spectrum. Our discussion of the task coordinator provides some insight into the higher levels of the spectrum. A task coordinator groups solutions and participants into processes in a manner that enables facilitators to work with knowledge that reflects the results created for a human institution. One might presume that a process organization would organize knowledge in a manner that facilitators could interpret as understanding that can be used to achieve objectives. This is only true to the extent that a human institution and its objectives can be captured in document models. A computer is a symbol processing device. Documents are a communication device: a system of information exchange in the form of symbols. Only those aspects of work and our institutions which are represented in document models as they are defined in this thesis can be part of the automation spectrum we have defined. The box for process organization is shown in the diagram with a dotted outline. The dotted line for it and other objects is used to indicate reservations about the degree to which the automation spectrum applies. The entrepreneur/director bubble represents work done by individuals that lead institutions defining objectives and shaping processes within a society. The reservations which apply to process organization also apply to their work.

At the lower levels, operating systems and computer hardware process signals in a manner that creates symbols which reflect data about the world. They provide the infrastructure which we us to create documents for communication. Figure 27 shows the degree of automation that can be achieved by software based on document models by locating it on the knowledge work automation spectrum. Not all software needs to be organized around document models, since not all software is intended for knowledge work automation. If other approaches to knowledge work automation are conceived they can be located on this spectrum

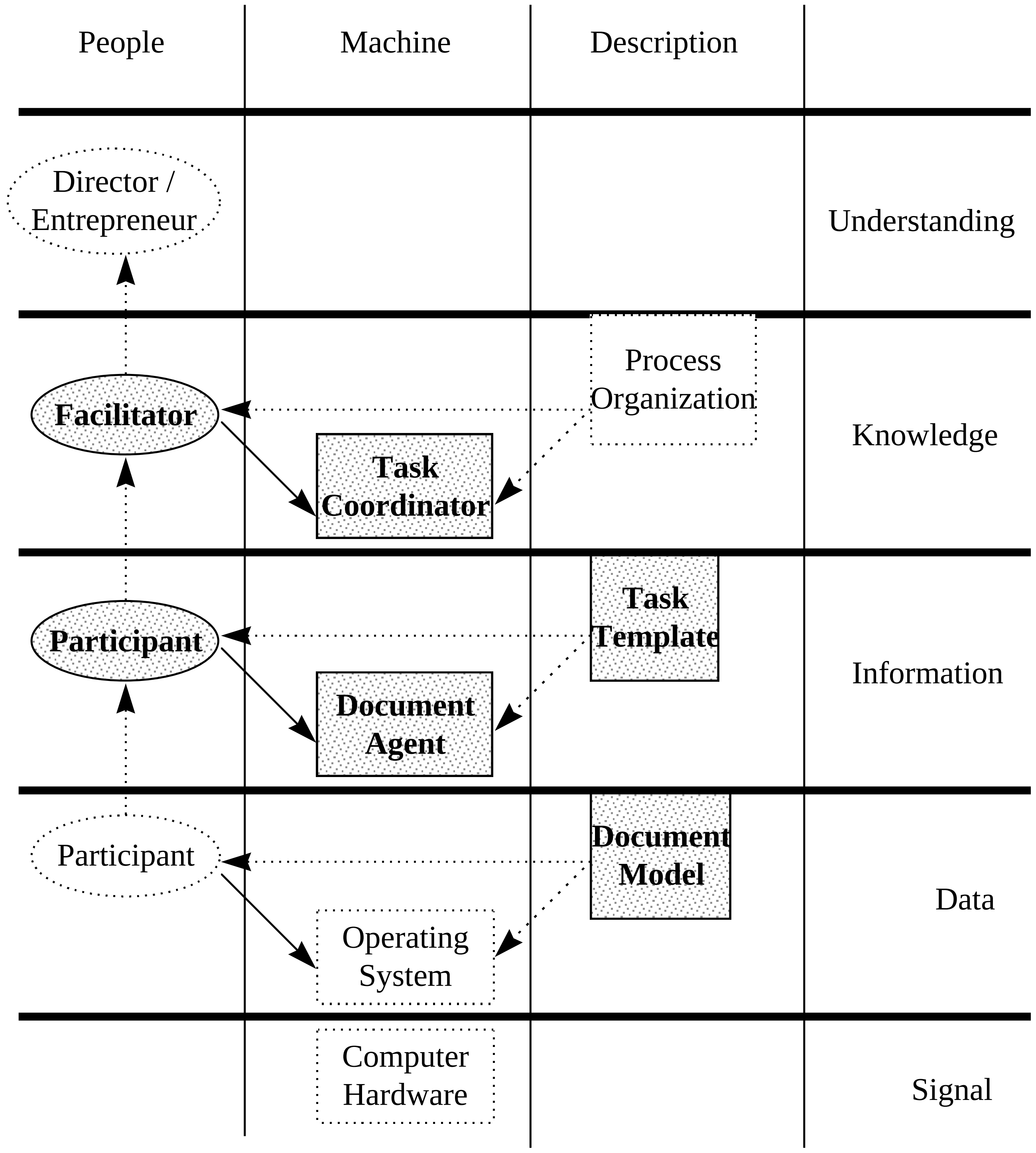


Figure 27. Knowledge Work Automation Spectrum

5.5.2 Benefits and Concerns

The benefits and concerns associated with automation hinge on the roles played by human participants and facilitators. Automation changes the manner in which work is accomplished. If automation can be used to improve the ability of human institutions to achieve their objectives, what will be the impact on participants and facilitators when it changes their work? It is also important to understand to what extent the benefits attainable by automation are dependent on the skills and contributions of participants and facilitators. To understand software automation requires an understanding of what people do as well as an understanding of what the technology does.

It is not necessarily appropriate or beneficial to approach a human institution from the point of view of a knowledge work process that can be automated by software. Software automation implies the standardization of the information exchanges which occur in knowledge work. Benefits will result from automation only to the degree that significant aspects of an institution can be viewed as standardized information exchanges. Those aspects must remain integrated with the institution once they are automated.

One of the initial benefits which result from automation is to alleviate the burden of work associated with data or symbol level interactions. In an electronic medium, there can be orders or magnitude improvements associated with data processing operations. Document agents with document models raise the level of interaction to an information based one. The implication is that the participants understand the meaning of the data with which they are working. Increased automation reduces the amount of work available for participants who do not understand the meaning of what they are working with.

When tasks are automated through task templates, then the burden of work associated with information level interactions are alleviated and order of magnitude improvements can be achieved. This does not necessarily reduce the amount of work for participants, but it does change the nature of the work. Less time is spent creating information. More time is spent choosing which information to create, and conveying the knowledge which that information contains. The quality and consistency of the knowledge conveyed by the information is improved. The complexity of task that can be managed by a person is also improved. Care must be taken that participants are knowledgeable enough to recognize when a template is not appropriate, and especially to recognize when a solution created by templates does not produce results as expected. Dramatic improvements in productivity and quality are leveraged by knowledgeable participants.

Productivity is one measurement of the results which help a human institution achieve objectives. To some degree, software automation is a means to replace participants or reduce the contribution of participants while maintaining or increasing productivity. For knowledge work, though, this view limits the benefits that can be obtained. Knowledge work communicates information that conveys knowledge to individuals in a manner that achieves the objectives of human institutions. The discovery of new information, new knowledge and new understandings of relevance creates new potentials for growth, productivity and objectives. Software technology "informates" [Zuboff 88] as well as automates. The feedback that is measured in automated reports provides new insights that can shape the work process. In a changing world, institutions and their work processes must adapt and evolve continually. In an automated work process, that evolution is only possible through the configuration of the process organization as well as the document models, document agents, and task templates which provide automation. Knowledgeable, skilled participants and facilitators provide the insight and the understanding which is used to define these. Without them, a knowledge work process is dead.

5.6 *Relationship To Other Work*

5.6.1 Research

Our understanding of the context for software automation within a human institution and its relationship to the overall work processes by which objectives for an institution are achieved has evolved through our experiences as a consultant with a number of organizations. In [Peyton 93], a list of guidelines is summarized that identify issues which need to be addressed in order to ensure success on automation projects. Many of the issues which need to be addressed are not primarily technical issues but are business and people-related. [Zuboff 88] provides an in-depth look at the business and social issues which are arising as we shift from “manual” and “paper-based” processes to “automatic” and “electronic” processes. New potential is also articulated in terms of the ability of software automation to *informate* a work process as well as *automate* it. Focusing software automation on supporting and coordinating knowledge work performed by different participants in a work process was first proposed in [Winograd & Flores 86]. The emphasis was on coordinating and structuring the communication between participants involved in a work process. It did not attempt to standardize or automate the performance of individual tasks as we have done in this thesis. [Stodin 91] discusses how to reorganize a work process to address more effectively the concerns of a client in a manner that leverages the capabilities of software automation technology to automate and informate.

5.6.2 Industrial Practice

There are an increasing number of software products like electronic mail, project management tools, group calendars etc. which are available for coordinating work processes. Work flow automation systems are used to automatically schedule and coordinate tasks and distribute the documents used amongst participants in a knowledge work process. They are usually based on electronic mail systems or groupware products like Lotus Notes. One of the first work flow automation products was the Coordinator system from Action Technologies which was based on the work in [Winograd & Flores 86]. There is also a growing market for business intelligence tools like Powerplay and Impromtu from Cognos which analyze and interpret information collected during a work process i.e. they focus on the potential of software to informate.

There are few off the shelf products currently that integrate coordination of tasks with the technology for automating tasks. There are a few niche products that provide some automated support for sales and contact management. Usually, though, integrated work automation must be custom built. In a number of domains, there are initiatives to define standards and establish the infrastructure which will make such products more practical. In both the CALS [Walter 92] initiative for technical documents and the TMN standards for telecommunications [Aidarous & Plevyak 94], document models and an agent architecture lay a foundation for transforming those domains of knowledge work into electronic domains. Standardization enables interoperability between different document agents for different aspects of the work process and interoperability between products from different vendors. That interoperability can facilitate the integration of task coordination and task automation.

Appendix

A. Projects

Our thesis has grown and matured through our participation in a number of projects at several institutions and companies. The following provides a brief description of these projects and our participation in reverse chronological order.

A.1 Department of Computer Science, Aalborg University

The Programming Languages Group at Aalborg University has shaped our understanding of software automation and provided the intellectual tools with which to express it. This thesis was written and defended at Aalborg University during 1995 and 1996. The Tilda project has been dedicated to the development of structured hypertext systems which support programming since 1989. Our participation in the project as a PhD student in 1990 laid the ground work for our understanding of software development in terms of documents which was applied in the SoDA and Axiant systems. Throughout the entire course of our research, discussions with the principal members of the group has been invaluable. They are:

Bent Bruun Kristensen
Kasper Osterbye
Kurt Normark

A.2 Knowledge Exchange Communication Ltd.

KEC is a business consulting firm with a proprietary methodology for organizing knowledge work processes around strategic objectives. From 1993-1995, as a senior consultant for The MirrorWorks Software Inc., we participated in a series of three pilot projects at large corporations in which software systems were built to automate knowledge work processes. In each case, a team of about 10 people was assembled, and several offices of 10 - 20 people used the new work processes. That experience crystallized the understanding of our thesis work and has shaped its presentation. The scenarios which we use to illustrate our architecture have been taken from these experiences. The MirrorWorks is now collaborating with KEC in the development of a software product for automation called ServicePlus Toolworks. The key participants we interacted with were:

Nick Stodin
Jamie Winterhalt
John Kambanis
Wayne Dustin

A.3 Bell Sygma Inc.

Bell Sygma is a telecommunications firm which provides services and products internationally as well as to its principal client, Bell Canada. From 1994-1995, as a senior consultant for The MirrorWorks Software Inc., we participated in the development of software systems to automate the provisioning and testing of circuits in a phone network. This experience introduced us the standards and architectures which are making automation possible in the telecommunications industry. A very large number of people were involved. Working with the following people has given us our understanding of automation in this domain:

Kevin Arbour	Rob Wolf	Andy Pospiech	Jamie Tholl	Steve Ardron
Guy Fortin	Lee Zhao	Steve Diteljan	Richard Ouellette	Yiwen Jiang
Alex Reid	Raja Hawa	Rob Walsworth	Wes McGregor	

A.4 Dataware Canada Inc.

Dataware Canada provides services and products for CD-ROM publishing and LAN-based full text retrieval databases. From 1993-1994, as product architect, we participated in the Intelligent Document Architecture Project which integrated a preliminary version of Dataware's Megatext 2.0 full-text retrieval product with systems for the automation of document publishing. In addition to the core team at Dataware, several departments of the Federal Government of Canada participated including the Department of National Defense which partially funded the project. This project introduced us to standards and initiatives (SGML, CALS) which address automation related to documents. Working with the following people has given us our understanding of automation in this domain:

Charlie Rabie	Arthur Gniazdowski	Tao Guan
Richard Boadway	Diane Montreuil	Mike Sweney
Henry Lewkowiec	Richard Brosseau	Andy Moffat

A.5 Cognos Inc.

Cognos produces application development tools and business intelligence products which it sells world wide. From 1991-1993, as a senior software engineer, we participated in the development of Axiant 1.0, a work group development product for building database systems. The approach to documents and document agents experimented with in SoDA was applied to Axiant and adapted to the requirements imposed by large scale industrial use. It was during the development of the Module Designer and Program Creation Assistants for Axiant 1.0 that the importance of task templates in our architecture was realized. A large number of individuals have been and continue to be involved with the Axiant product. My interaction with the following people contributed to the understanding that is reflected in this thesis.

Ron Zambonini	Brad Cameron	Stewart Winter	Bob Deskin
Gary Puckering	Celine Goyette	Mychelle Mollot	Chris Bowering
Colin Moden	Al Hendrickson	Mike Baggot	Pat Froese-Germaine
Hal O'Connell			

A.6 Kestrel Institute and Knowledge Systems Laboratory, Stanford University

The Kestrel Institute is a private research laboratory specializing in combining artificial intelligence and formal methods of software specification and development. The Knowledge Systems Laboratory is part of the Department of Computer Science, and specializes in the development of expert systems technology and medical information systems. As a PhD student from 1987-1991, we participated in a joint project between the two organizations called the Knowledge Assisted Software Engineering project. It was within the context of this project that we built the SoDA system in which the initial architecture of our thesis was developed and experimented with. The following people participated and influenced the course of my work.

Ed Feigenbaum	Raymonde Guindon	Steve Wesfold	Jorge Phillips
Cordell Green	Nelleke Aiello	Tom Pressburger	Jim Rice
Penny Nii	Raul Duran	Doug Smith	

A.7 Center for the Study of Language and Information, Stanford University

The Center for the Study of Language and Information promotes research in the departments of computer science, linguistics, philosophy, and psychology which addresses the nature of communication and the manner in which information is processed. As a PhD student from 1985-1987, we were a member of the System Development Languages Group which developed a syntax-based environment in which to experiment with languages for specifying software systems. The key members of the group were:

Terry Winograd	Mary Holstege	Raul Duran
Kurt Normark	Kasper Osterbye	

Bibliography

[Aidarous & Plevyak 94] S. Aidarous and T. Plevyak (editors), *Telecommunication Network Management into the 21st Century*. IEEE Press, Piscotavy, NJ, US, 1994.

[Amarel 68] S. Amarel. "On representation of problems of reasoning about actions". *Machine Intelligence 3*, pp. 131-171. (ed. Michie, D.) Edinburgh: Edinburgh University Press.

[Balzar et al 83] R. Balzar, T. Cheatham, C. Green. "Software Technology in the 1990's: Using a new paradigm," *Computer*, 1983.

[Black et al 87] A. Black, N. Hutchinson, E. Jul, H. Levy, L. Carter, "Distribution and Abstract Types in Emerald". IEEE Transactions on Software Engineering, vol. SE13, no. 1, pp. 65-76, Jan.1987.

[Booch 94] G. Booch. *Object-Oriented Analysis and Design*. Benjamin/Cummings, Redwood City, CA, 1994. Second Edition.

[Brooks 86] F. Brooks. "No Silver Bullet - Essence and Accidents of Software Engineering", in *Information Processing 86*. H.Kugler (Ed.), Elsevier Science Publishers B.V. (North-Holland) 1986.

[Brooks 95] F. Brooks. *The Mythical Man-Month: Anniversary Edition.* 1995, Addison Wesley.

[Brown 91] A. Brown. *Integrated Project Support Environments: The Aspect Project*. Academic Press Limited, London, England, 1991.

[Buchannon 82] B. Buchanon, "New Research on expert systems," (eds. Hayes, Michie, and Pao), *Machine Intelligence 10*, pp-269-299, Ellis Horwood 1982.

[Campbell 88] I. Campbell. "Emeraude - A Portable Common Tool Environment". *Information and Software Technology* 30(4), May 1988.

[CCITT 92a] Principles for a Telecommunication Management Network. CCITT Recommendation M.3010, International Telephone and Telegraph Consultative Committee, 1992.

[CCITT 92b] Generic Network Information Model. CCITT Recommendation M.3100, International Telephone and Telegraph Consultative Committee, 1992.

[Chen 77] P.Chen, The entity-relationship approach to logical database design. Q.E.D. Information Sciences, Wellesley, Massachusetts, 1977.

[Coad & Yourdon 90] P. Coad, E. Yourdon. *OOA - Object Oriented Analysis*. Yourdon Press, Prentice Hall, Englewood Cliffs, 1990, 1991.

[Davis et al 77] R. Davis, B. Buchannon, E. Shortcliffe. "Production rules as a representation of a knowledge-based consultation program". *Artificial Intelligence 8*, pp15-45, 1977.

[Gamma et al 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*, Addison -Wesley Reading Massachusetts, 1995

[Goldberg and Robson 94] A. Goldberg, D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison Wesley, 1984.

[Goldberg 86] Allen T. Goldberg, "Knowledge-based Programming: A Survey of Program Design and Construction Techniques", *IEEE Trans. on Software Eng.*, Vol. SE-12, pp752-768, July 1986.

[Goldfarb & Rubinsky 90] C. Goldfarb, Y. Rubinsky. *The SGML Handbook*, Clarendon Press, Oxford, England, 1990.

[Guindon 90] R. Guindon, “Designing the design process: Exploiting opportunistic thoughts”. *Human-Computer Interaction*, Volume 5. 1990

[Guindon 90] R. Guindon, “Knowledge exploited by experts during software system design.” *International Journal of Man-Machine Studies*. vol 33, pp279-304, 1990.

[Haanlykken & Nygaard 81] P. Handlykken, K. Nygaard: “The DELTA System Description Language: Motivation, Main Concepts and Experience from use”. In: *Software Engineering Environments* (ed. H. Hunke), GMD, North-Holland, 1981.

[Jackson 83] M. Jackson: *System Development*. Prentice Hall, 1983.

[Jacobsen et al 92] I. Jacobsen, M. Christerson, P. Jonsson, G. Overgaard. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, Wokingham, England, 1992.

[Johnson 87] W. Johnson. “Turning Ideas Into Specifications”. *ACM TOPLAS*, April 1987.

[Kristensen 94] B.B.Kristensen. “Complex Associations: Abstractions in Object-Oriented Modeling”. *Proceedings of OOPSLA'94*, 1994.

[Kristensen & Osterbye 94] B. B. Kristensen, K. Oesterbye. “Conceptual Modeling and Programming Languages”. *SIGPLAN Notices*, 29(9):81-90, September 1994.

[Madsen et al 93] Ole Lehrmann Madsen, Birger Moeller-Pedersen, Kristen Nygaard.   *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley Publishing Company, Wokingham, England 1993.

[Martin 85] James Martin. *Fourth Generation Languages*, volume 1. Prentice Hall, Englewood Cliffs, 1985.

[Martin 89] James Martin. *Information Engineering,* Prentice Hall, Englewood Cliffs, 1989.

[Mathiessen et al] L. Mathiessen, A. Munk-Madsen, P. Nielsen, J. Stage, “Modeling Events in Object-Oriented Analysis”. pp 88-104 (Eds. D.Patel, Y. Sun, S. Patel) *1994 International Conference on Object-Oriented Information Systems*, Springer-Verlag, London, 1995.

[Mathiessen et al 95] L. Mathiessen, A. Munk-Madsen, P. Nielsen, J. Stage, *Objektorienteret Design*. Forlaget Marko, Aalborg. (In Danish) 1995.

[Nanard et al 93] Nanard M. et al., “The Generalist Metaphor: Macroscopic Knowledge Acquisition and Use on a Technical Document Base”, *Proceedings JAVA ‘93,* St. Raphael (France), April 1993.

[Nanard & Nanard 93] J. Nanard, M. Nanard, “ Should Anchors Be Typed Too? An Experiment with MacWeb”, *Fifth ACM Conference on Hypertext Proceedings*, Seattle, Washington. November 1993.

[Nii & Aiello 79] H. Nii, N. Aiello. “AGE (Attempt to Generalize): A knowledge-based program for building knowledge-based programs”. Proceedings IJCAI-79, pp.645-655. 1979.

[Normark 87] K. Normark, *“Transformations and Abstract Presentations in a Language Development Environment”*. PhD Thesis, Technical Report DAIMI PB-222, The Computer Science Department, Aarhus University, Denmark, February 1987.

[Normark 89] K. Normark, *“Programming Environments - Concepts, Architectures and Tools”*, Technical Report R-89-55, Institute of Electronic Systems, Aalborg University, Denmark,1989.

[Normark 89b] K. Normark, *“Open Points and General Programs”*, Technical Report R-89-3, Institute of Electronic Systems, Aalborg University, Denmark,1989.

[Nygaard & Dahl 81] K. Nygaard, O.J. Dahl: “Simula 67”. In: R. W. Wexelblat (ed.), *History of Programming Languages*, 1986.

[Orfali et al 96] R. Orfali, D. Harkey, J. Edwards. The Essential Distributed Objects Survival Guide. John Wiley & Sons, Inc., New York, 1996.

[Osterbye 89] K. Osterbye, “Implementation Support for the Specification Language Aleph.”, Technical Report R-89-18, Phd Thesis. Institute for Electronic Systems. Aalborg University, Denmark,1989.

[Osterbye 90] K. Osterbye, “Parts, Wholes, and Subclasses”. in *Proceedings of the 1990 European Simulation Conference*, ed. B. Shmidt, pp259-263, 1990.

[Osterbye & Normark 93] K. Osterbye, K. Normark. “The Vision and the Work in the HyperPro Project.” Technical Report R-93-2012. Institute for Electronic Systems. Aalborg University, Denmark April 1993

[Osterbye 93] K. Osterbye. “Literate Smalltalk Programming using Hypertext”. Technical Report R-93-2025. Institute for Electronic Systems. Aalborg University, Denmark August 1993.

[Perry 87] D. Perry, “Software Interconnection Models”, *Ninth International Conference on Software Engineering,* IEEE Computer Society Press, 1987

[Peyton 87] L. Peyton, “A General Presentation Facility for Language Design Environments”, Technical Report IN-CSLI-87-10, Stanford University, California, 1987.

[Peyton 88] L. Peyton, “A Transformation Approach to Software Redesign”, Technical Report KSL-88-37, Stanford University, California, 1988.

[Peyton et al 90] L. Peyton, M. Gersh, G. Swietek, “Impact of Knowledge-based Software Engineering on Aerospace Systems”, *Aerospace & Software Engineering*, C. Anderson, Ed., AIAA 1990.

[Peyton 93] L. Peyton, “Evaluation Criteria for Expert Systems Projects”, Technical Report TMW-93-01, The MirrorWorks Software Inc., Ottawa, Canada. 1993.

[Philips 83] J. Phillips, “Self-Described Programming Environments”. Phd Thesis. Technical Report STAN-CS-84-1008, Stanford University, California, 1983.

[Reps 84] T. Reps, *Generating Language-Based Environments,* MIT Press, Cambridge, MA 1984.

[Rich 81] C. Rich, “A formal representation for plans in the programmer’s apprentice”. *IJCAI 7,* 1044-1052. 1981

[Rumbaugh et al 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-Oriented Modeling and Design*. Prentice-Hall 1991.

[SGML 86] ISO 8879-1986 Standard Generalized Markup Language

[Shlaer & Mellor 88] S. Shlaer & S.J. Mellor: *Object-Oriented Systems Analysis - Modeling the World in Data*. Yourdon Press, Prentice Hall, 1988.

[Slagle & Wick 88] J. Slagle and M. Wick. “A Method for Evaluating Candidate Expert System Applications”. *AI Magazine,* pp 44-53, Winter 1988.

[Smith, Kotik, & Westfold 85] D. Smith, G.B. Kotik, and S.J. Westfold, “Research on knowledge-based Software Environments at Kestrel Institute,” *IEEE Trans. Software Eng*., vol. SE-11, pp. 1278-1295, Nov. 1985

[Smith & Lowry 89] D. Smith, M. Lowry. “Algorithm Theories and Design Tactics. In *Proceedings of the International Conference on Mathematics of Program Construction. LNCS 375*, pp 379-398, L. van de Snepscheut, Ed., Springer-Verlalg, Berlin, 1989.

[Smith 89] D. Smith. “KIDS: A Semi-Automatic Program Development System”, Technical Report KESTREL-10-89, Kestrel Institute, Palo Alto, California, 1989.

[Stallings 93] W. Stallings, *SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards*. Addison-Wesley Publishing Co. 1993.

[Stodin 91] N. Stodin, “The Clientship Strategy”, Technical Report KEC-91-01, Knowledge Exchange Communication Ltd., Toronto, Canada, December 1991.

[Thomann 94] J. Thomann, “Data Modeling in an OO World The Potential Problems of Using Inheritance Hierarchies”. *American Programmer,* Vol.7, No. 10, pp44-53, October 1994.

[Walter 92] M. Walter, “A Look Inside JCALS: Integrated Approach to Technical Manuals”, *The Seybold Report on Publishing Systems,* Vol. 21 N.11, February 1992.

[Waterman 86] D. Waterman, *A Guide to Expert Systems*. Addison-Weslely, 1989.

[Waters 86] R. Waters. “KBEmacs: Wheres the AI?”, *The AI Magazine*. Spring 1986.

[Webster 87] *Webster’s Ninth New Collegiate Dictionary*. Springfield Ma., USA: Merriam Webster Inc. 1987

[Winograd 75] T. Winograd. “Frame representations and the procedural-declarative controversy.” In D. Boborow & A. Collins (Eds.), *Representation and Understanding: Studies in Cognitive Science*. New York: Academic Press, 1975

[Winograd & Flores 86] T. Winograd, F. Flores. *Understanding Computers and Cognition: A New Foundation for Design*. Ablex. Norwood, NJ 1986

[Winograd 86] T. Winograd, “A language/action perspective on the design of cooperative work”, *Proceedings of Conference on Computer-Supported Cooperative Work,* Austin, TX, 1986.

[Winograd 87] T. Winograd. “Muir: A tool for language design”. Technical Report CSLI-87-81, Center for the study of Language and Information. Stanford University. March 1987.

[Woods 75] W.A. Woods. “What’s in a link: Foundations for semantic networks”. In D.G. Bobrow and A. Collins, editors, *Representation and Understanding, Studies in Cognitive Science*. Academic Press, New York, 1975.

[Wozniewicsz 94] A. Wozniewicsz, “Component Wars: A Perspective on Objects”. *American Programmer,* Vol.7, No. 10, pp23-27, October 1994.

[Zuboff 88] S. Zuboff, *In the Age of the Smart Machine: The Future of Work and Power*. Basic Books Inc., New York, 1988.