Ubik: A Framework for the Development of Distributed Organizations

by

Stephen Peter de Jong

B.S. Physics, Pennsylvania State University 1962
M.S. Operations Research, New York University 1968

Submitted to the
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1989

© Massachusetts Institute of Technology 1989. All rights reserved

Signature of Author ....
Department of Electrical Engineering and Computer Science
11 August 1989

Certified by ....
Carl E. Hewitt
Associate Professor, Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith, Chairman
Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

DEC 27 1989

LIBRARIES

ARCHIVES

Ubik: A Framework for the Development of Distributed Organizations

by

Stephen Peter de Jong

Submitted to the
Department of Electrical Engineering and Computer Science
on 11 August 1989 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy in
Artificial Intelligence

Abstract

An organization in Ubik consists of interrelated applications, distributed over multiple locations, executing in parallel. Applications are built out of the Ubik basic computational object called a configurator. Configurators represent both organizational structure and action. Structure is represented by a linked network of configurators. Action is carried out by the configurators passing messages between themselves. The configurator networks can be distributed over multiple computers. Special configurators called constructors, tapeworms, and questers build, maintain, and reason over the distributed networks.

Ubik has a high-level language which can be used by end-users to describe their applications in such a way such that there is a close correspondence between their mental model of the application and the Ubik representation of it. This correspondence eases both the implementation and maintenance of the applications.

Organizational power is the process by which an organization focuses its limited resources to accomplish its most important goals. In Ubik, power refers to the competition between the parallel executing configurators for the limited computational resources. Ubik's configurators compete with each other for computational power in such a way that the organizational goals can be adequately met.

Organizational development refers to the continual evolution necessary in a large organization to cope with the changing external environment. Ubik can monitor its actions and reason over its structure; these abilities allow Ubik to change its representation so that the internal computer model more closely matches the needs of the external organization.

Ubik in its current stage of development is a framework for an organizational development system. Much more design and implementation work needs to be done to fully work out the Ubik concepts.

Thesis Supervisor: Carl Hewitt
Title: Associate Professor of Computer Science and Engineering

Acknowledgements[1]

This research was carried out in the Message Passing Semantics group, within the Artificial Intelligence Laboratory at MIT. The group was founded by Carl Hewitt to pursue work in object-oriented programming, parallelism, distributed computation, and organizational semantics. This thesis was strongly influenced by the ideas generated by the group over many years. I would particularly like to thank the following current and former members of the group: Gul Agha, Jonathan Amsterdam, Henry Lieberman, Carl Manning, and Thomas Reinhardt. Carl Manning provided detailed comments on an earlier draft of the thesis, for which I'm very thankful. David Kirsh, a member of the Artificial Intelligence Laboratory, provided helpful suggestions when critiquing an earlier Ubik paper. His suggestions helped improve the content and presentation of this thesis.

I would like to thank my thesis committee—Carl Hewitt, Marvin Minsky, and Peter Szolovits—for encouraging this work. I would like to thank Carl Hewitt, my thesis advisor, for creating the environment in which this work was pursued. Carl's work on Actors and our joint work on Open Systems provided important technical background for this thesis. I have known and worked with Carl for many years. I thank him for his long-time support. Marvin Minsky encouraged me to pursue research in organizations and bureaucracies. His work on *The Society of Mind* provided a foundation for much of the thinking that went into Ubik. I would particularly like to thank Peter Szolovits for his detailed discussions on the technical aspects and presentation of this thesis. His help lead to a great improvement in the thesis.

Much of the funding for this work came from the System Development Foundation. I would like to thank Charles Smith, its program director, for his generous support and encouragement.

Jim Gray, a colleague and friend for many years, continually encouraged me to pursue this work. Technical discussions with Jim, on data base and operating systems, has kept me in touch with the world outside of MIT. Sailing with Jim on San Francisco Bay has kept the thesis in perspective.

Finally I need to thank my family for their many years of support, while I was a student at MIT. Norma, my wife, carefully edited the many drafts of this thesis. She kept the household going while I was otherwise preoccupied. I dedicate the thesis to her. Deborah, my youngest daughter, is eight years old. Since I have been at MIT for eight years, she thinks having a father as a student is natural. My son, Stuart, and daughter, Sandy, are much older; they know having a father as a student is unnatural. Stuart graduated from high school and college, and Sandy from grammar school and high school, while I have been at MIT. I thank them all for their support and sacrifices. Lastly I would like to thank my parents, who have kept a close watch on my educational progress from Florida. Their help and encouragement has been greatly appreciated.

[1]This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Systems Development Foundation and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract NE00014-85-K-0124.

Contents

1 **Introduction** 10

2 **Organizations** 13
2.1 Bureaucracy 14
2.2 Typologies 15
2.3 Organizations and their Environment 16

3 **Applications** 18
3.1 Purchasing Organization 19
3.1.1 Gradual Automation 23
3.1.2 End-user Programming 24
3.1.3 Bureaucracy 27
3.1.4 Batching 31
3.1.5 Regrouping 34
3.1.6 Questers 38
3.1.7 Parasitic Tapeworm 41
3.1.8 Freedom of Action Tapeworm 44
3.2 Development of Large Software Systems 47
3.2.1 Version Dependency Sub-organization 47
3.2.2 Functional Management Sub-organization 48
3.2.3 Project Management Sub-organization 49
3.2.4 Organizational Model 49

4 **Organizational Structure** 53
4.1 Configurators 54
4.2 Constructors 57
4.3 Tapeworms 58
4.4 Prototypes 60
4.5 Questers 61
4.6 Distribution 63
4.7 Organizational Concepts 66

5 **Action** 68
5.1 Message Passing 69
5.2 Or Parallelism 72

5.3 And Batching 73
5.4 Bureaucratic Paths 74
5.5 Serializers 75

6 Tapeworms **77**
6.1 Installation 79
6.2 Types 81
6.3 Operations 81
6.4 Tapeworm and Quester Examples 82
6.4.1 Freedom of Action Tapeworms 82
6.4.2 Parasitic Tapeworm 86
6.4.3 Self-propagating Quester 88

7 Power **90**
7.1 Sponsor Structure 94
7.2 Interacting Sponsors 97

8 Development **98**
8.1 Message Elimination 100
8.2 Regrouping 103
8.3 Reclustering 106
8.4 Regression 107
8.5 Prototype Development 108
8.5.1 Forming Collections 108
8.5.2 Creating a Prototype from a Collection 108
8.5.3 Finding a Prototype 109
8.6 Bureaucratic Development 110

9 Conclusion **112**

A Early Ubik and Its Implementation **114**
A.1 Operations 118
A.2 Configurator Unification 119
A.3 Application Examples 120
A.3.1 Building a Network 120
A.3.2 Message Sending and Receiving 121
A.3.3 Unifying with the Configurator Body 122
A.3.4 Configurator Variables 124
A.3.5 Distributed Message Send 129
A.3.6 Data Flow 132
A.3.7 More than Manager Query 135
A.3.8 Batching 136
A.3.9 Tapeworms 138
A.4 Implementation 140
A.4.1 Basic Objects 141
A.4.2 Front-end 144

A.4.3 Evaluator 144
A.4.4 Model and Networks 154
A.4.5 Tapeworm 157
A.4.6 Unification 159
A.4.7 Utilities 164
A.5 Conclusion 165

B Related Work **166**
B.1 Hypertext 167
B.2 Relational Databases 177
B.3 Rule and Frame Based Systems 187
B.4 Semantic Nets 199

Bibliography **206**

List of Figures

3.1 Ubik Configurator Format 19
3.2 Purchase Organization 21
3.3 Purchase Organization Forms 22
3.4 Purchase Requisition Interactive Dialog 23
3.5 End-user Written Purchase Requisition Dialog 25
3.6 Flow of Purchase-Requisition 26
3.7 Bureaucratic Decision Making in the Purchasing Department 28
3.8 Buyer Creating Purchase-order 29
3.9 Flow of Purchase-order 30
3.10 Batching of Shipping Department Forms 31
3.11 End-user Written Program to Batch Forms 32
3.12 Flow of Notification-of-receipt Form 32
3.13 Batching of Purchase-orders at the Billing Configurator . . . 33
3.14 Creating an Invoice from Multiple Purchase-orders 35
3.15 End-user Written Program to Create an Invoice 36
3.16 Flow of Invoice 37
3.17 Quester to Find Purchasers of a Vendor's Products 39
3.18 Quester to Find Departments which Accept Purchase-orders . 40
3.19 Parasitic Tapeworm on Purchase-order 42
3.20 Installation of Parasitic Tapeworm 43
3.21 Freedom of Action Tapeworm 45
3.22 Installation of Freedom of Action Tapeworm 46
3.23 Version Dependency Sub-organization 48
3.24 Functional Management Sub-organization 49
3.25 Project Management Sub-organization 50
3.26 Software Development Organization 51
3.27 Software Development Organization Forms 52

4.1 Ubik Configurator and Link Types 55
4.2 Frame and Ubik Representation of an Employee 56
4.3 Constructor Configurators 57
4.4 Tapeworm Paycycle Censor 58
4.5 Tapeworm Salary Censor 59
4.6 Employee Prototype Hierarchy 60
4.7 Labeled Links 61

4.8 Quester Over Labeled Links 62
4.9 Distributed Configurator 63
4.10 Message Passing between Distributed Configurators 64
4.11 Linking between Models 64
4.12 Context and Individuals 65

5.1 Message Passing 69
5.2 Flow Links 69
5.3 Reply Messages 70
5.4 Sending to Multiple Destinations 71
5.5 Or Parallelism 72
5.6 And Batching 73
5.7 Bureaucratic Paths 74
5.8 Bank Account with Serializer 76

6.1 Tapeworm 78
6.2 Installing Tapeworms 79
6.3 Commutative Tapeworm 80
6.4 Freedom of Action Tapeworm Installed in the Purchasing Department 84
6.5 Freedom of Action Tapeworm 85
6.6 Parasitic Tapeworm 87
6.7 Self-propagating Quester 89

7.1 Sponsor with Linked Configurators 92
7.2 Centralized Sponsor Control 94
7.3 Decentralized Sponsor Control 95
7.4 Coordinated Sponsor Control 96
7.5 Sponsor Interaction 97

8.1 Blanket Purchase Orders 101
8.2 Purchase Order Draft 102
8.3 Regrouping 105
8.4 Reclustering 106
8.5 Regression 107

A.1 Early Ubik Two-Dimensional Notation 115
A.2 Organization Network 117
A.3 Message Sending 121
A.4 Unifying with the Configurator Body 123
A.5 Configurator Variables 125
A.6 Parallel *And Expression* 130
A.7 Parallel *And Expression* Flow 131
A.8 Data Flow 133
A.9 Batching Input Messages 136
A.10 When-modified Tapeworm 138

B.1 Nodes in the Notecard Hypertext System 169
B.2 Topic Browser for Notecard Hypertext System 170
B.3 KMS Distributed Hypertext System 170
B.4 Intermedia Webs 171
B.5 Intermedia Global Browser without a Focus of Attention . . . 173
B.6 Reducing Links Displayed with Subtree Detail Suppression . 174
B.7 Result of Frisse's Principles in Calculating Promising Browsing Paths 176
B.8 Paradox Query 177
B.9 Paradox Join 178
B.10 Paradox Form for Viewing a Row in a Table 179
B.11 Paradox Form for Viewing Two Tables 180
B.12 Paradox Format for Printing a Report 181
B.13 Paradox Report 182
B.14 Paradox Bar Graph using a Crosstab table 183
B.15 Units 188
B.16 Automotive Unit 189
B.17 Kee Class Structure 190
B.18 KEE Multiple Inheritance 190
B.19 Rules and Units 191
B.20 KEE Worlds and Versions 192
B.21 KEE Active Images 192
B.22 KEEconnection 194
B.23 Rules and Tapeworms 196
B.24 Kl-one Net 201
B.25 Ubik Network 201
B.26 Krypton and Kl-two 202
B.27 Kl-one Classification 203
B.28 Ubik Statement of Kl-one Classification 204

Chapter 1

Introduction

Ubik is a system for building multiple, interrelated computer applications. The applications are distributed over computer networks and can execute on parallel computers. An organization in Ubik is a collection of applications, and the end-users who interact with the applications. Ubik explores the following issues in the construction, execution, and maintenance of organizations:

1. Development of a high-level, object-oriented language to support interrelated, distributed, and parallel applications.
2. Represent an organization external to Ubik, such that there is a close correspondence between the structure and activities of the external organization and the structure and activities of the Ubik representation.
3. Explore the interaction between an organization's structure and action. The structure is represented by a semantic net consisting of message passing objects. The action consists of messages between the objects.
4. Use the Ubik representation for the high-level definition of new applications by the people within the organization who perform these applications.
5. Support the gradual automation of a business organization where the Ubik control of the organization coexists with the manual control and activity of the organization.
6. Maintain the power relationships between the cooperating and competing applications. Organizational power is the process by which an organization focuses its limited resources to accomplish its most important goals.
7. Automatically develop new Ubik representations to more closely match the continually changing business organization.

Organizations are represented within Ubik as open, distributed, and parallel systems.

An organization is open in that it is a continually changing entity of uncertain scope [38,39,41,42]. The closed world assumption, which is used by most computer systems, states that the failure to find a goal within the system is equivalent to the goal not existing. This assumption is not valid for open systems. The finding of a goal within an open system is usually a function of the amount of organizational resources committed to the search. Within Ubik these resources are specified by the use of sponsors, as described in chapter 7. Even if a goal is not completely found within Ubik, the search can return partial information concerning the goal which can be used for taking further organizational action.

An organization is a distributed collection of subparts. Each subpart has its own goals, concepts, and action. Communication is possible within an organization to the extent that the subparts share common goals and concepts. Some of the cost of communications is in establishing and maintaining the goals and concepts. The independent action of the subparts leads to both organizational conflict and robust behavior. The conflict comes from taking action on conflicting goals; the robustness comes from the redundancy of overlapping actions. The issues of communications and conflict are discussed throughout this thesis.

An organization operates in parallel. One of the prime purposes of an organization is to marshall enough resources and support enough parallel activity to accomplish a goal in a timely manner. Ubik objects naturally execute in parallel. Many of the mechanisms within Ubik are to coordinate this parallel activity.

Ubik views an organization as multiple collections of overlapping suborganizations, where each suborganization is a collection of suborganizations, people, activities, and materials. The organization is continually changing. These changes are the result of the following:

- Changing environmental conditions. These changes can include new economic conditions, material availability, personnel availability, and customer demand.

- Accidental and planned changes in the organization structure and activities. These changes can include new applications, new product areas, changes in policies, and personnel changes.

- Internal power conflicts. These changes result from the internal competition within an organization which arises when the organization, with limited resources, must satisfy multiple organizational goals.

Computerized application systems need constant maintenance. This maintenance attempts to match the computerized system to the changing

organizational conditions. Failure to maintain a computer application system results in it either becoming irrelevant or a hindrance to the operation of the organization.

To meet the goals of the Ubik system and the view of an organization as a continually changing entity, Ubik is built in a coherent way out of collections of objects called configurators. Collections of configurators form an organization. These collections represent the organization, perform organizational activity, and continually reorganize themselves. Chapter 2, on organizations, discusses various models of organizations. Chapter 3, on applications, describes the use of Ubik in writing and executing applications. Chapter 4, on organizational structure, describes how organizational representations are built out of configurators. Chapter 5, on action, describes how configurators perform organizational action. Chapter 6, on tapeworms, shows how configurators can be used to monitor, censor, and reason over Ubik organizations. Chapter 7, on power, describes how the organization's various suborganizations and activities compete for the limited computational resources. Chapter 8, on development, describes the ways in which organizations, built out of configurators, can be reorganized. Appendix A, on implementation, describes the implementation of an early version of Ubik. Appendix B, on related work, discusses the relationship of Ubik to hypertext, relational database, knowledge base, and semantic net systems.

The name Ubik was taken from a novel by Philip K. Dick [29]. Ubik stands for *ubiquity* which is defined as follows [34]:

> the state, fact, or capacity of being, or seeming to be, everywhere at the same time; omnipresent.

Much of the complication of organizing is that an organization needs to be everywhere, but can't be everywhere at the same time. This thesis deals with the mechanisms needed to design computer applications around this limitation.

Ubik in its current stage of development is a framework for an organizational development system. Much more design and implementation work needs to be done to fully work out the Ubik concepts.

Chapter 2

Organizations

Numbers, the fourth book of the Old Testament, describes the establishing of an organization for the people of Israel. The book starts out with a census of the twelve tribes of Israel. It then describes the structure of their land and housing, which at this nomadic state of their development is a temporary campground. During the book the structure of the priesthood, the judicial system, the army, and the interpersonal relationships are established. In addition, the relationship of the Israelite organization to the environment evolves. The environment consists of the following:

- Food supply - Much of their food comes from gathering manna, hunting quail, and finding grazing land for their cattle. Their cattle are the only food supply within the organization.
- Land over which they pass - Some of this land is owned by other tribes, whom they must fight for passage.
- Other cultures - They must keep separate from the surrounding tribes in order to avoid being absorbed into them.

The relationship between the Israelite organization and its environment keeps changing. When the Israelites settle in Israel, the first two environmental items, the food supply and land, become part of the organization. The third item, the relationship to other cultures, is formalized.

The central organizational theme of Numbers is that the organization is in dynamic flux. No sooner has one problem been solved, then another or the same problem arises again. Every once in a while a new organizational entity is created, and new authority relationships are established. Some of these last for a section, some for a chapter, and some still exist. The insight into the eternal nature of organizational conflicts is one of the reason the Bible is still relevant. In fact, the organizational conflicts of Numbers is currently more relevant than the actual organization described.

2.1 Bureaucracy

Weber [64] specifies the concept of an organization as follows:

> In the field of economically oriented action, 'organization' is the technical category which designates the ways in which various types of services are continuously combined with each other and with non-human means of production... A profit-making organization is spoken of wherever there is continuous permanent coordinated action on the part of an entrepreneur. Such action is in fact unthinkable without an 'organization', though, in the limiting case, it may be merely the organization of his own activity, without any help from others... But in a market economy, it is possible for a number of technically separate organizations or 'plants' to be combined in a single enterprise. The latter receives its unity by no means alone through the personal relationships of the various units to the same entrepreneur, but by virtue of the fact that they are all controlled in terms of some kind of consistent plan in their exploitation for purposes of profit...

Weber, in the book *The Theory of Social and Economic Organization*, creates some of the basic concepts of sociology in describing organizational economic action, authority, and co-ordination. His emphasis is on the interaction of people which permits the creation of organizations. His concept of a bureaucratic administration comes closest to what today would be a computer organization:

> Experience tends universally to show that the purely bureaucratic type of administrative organization—that is, the monocratic variety of bureaucracy—is, from a purely technical point of view, capable of attaining the highest degree of efficiency and is in this sense formally the most rational known means of carrying out imperative control over human beings. It is superior to any other form in precision, in stability, in the stringency of its discipline, and in its reliability. It thus makes possible a particularly high degree of calculability of results for the heads of the organization and for those acting in relation to it. It is finally superior both in intensive efficiency and in the scope of its operation, and is formally capable of application to all kinds of administrative tasks.
>
> The development of the modern form of the organization of corporate groups in all fields is nothing less than identical with the development and continual spread of bureaucratic administration. This is true of church and state, of armies, political parties, economic enterprises, organizations to promote all kinds

> of causes, private associations, clubs, and many others. Its development is, to take the most striking case, the most crucial phenomenon of the modern Western state. However many forms there may be which do not appear to fit this pattern, such as collegial representative bodies, parliamentary committees, soviets, honorary officers, lay judges, and what not, and however much people may complain about the 'evils of bureaucracy,' it would be sheer illusion to think for a moment that continuous administrative work can be carried out in any field except by means of officials working in offices. The whole pattern of everyday life is cut to fit this framework. For bureaucratic administration is, other things being equal, always, from a formal, technical point of view, the most rational type. For the needs of mass administration to-day, it is completely indispensable. The choice is only that between bureaucracy and dilletantism in the field of administration.

A goal of Ubik is to build a computer bureaucracy. Weber's praise of bureaucracy is gratifying. Most reactions to this goal of Ubik would be quite negative, since it involves putting all the rigidity of the worst type of human organizations into a computer. The negative reaction is somewhat justified from Weber's static notion of a bureaucracy. The biblical notion of a dynamic, continually tested, and changing organization is a better model for a bureaucracy then Weber's.

Marvin Minsky's *The Society of Mind* [53] describes how a computer organization consisting of multiple special parts can be intelligent. His model is both bureaucratic and dynamic. New organizational structures are continually created to supplement and, to a limited extent, replace the existing structures. In his theory, a function which a human obtains early in life cannot be replaced, because many other functions are built using it. So instead of replacement, there is bureaucratic organization. Bureaucratic organization creates decision making layers which decide under what conditions the underlying functions can be used. This is summed up by Minsky in Papert's Principle:

> The hypothesis that many steps in mental growth are based less on the acquisition of new skills than on building new administrative systems for managing already established abilities.

2.2 Typologies

Classifications in sociology are called typologies. Organizational typologies, according to Scott [59], *are relatively weak and nonpredictive because by their nature social organizations are open systems—highly permeated by and interdependent with their environments and comprised of loosely linked and*

semiautonomous component systems. Even though weak and nonpredictive, typological analysis is a useful technique to gain insight into organizations. Scott sums up the three views of organization discussed previously (bureaucratic, continually changing, and interacting with the environment) with the following typology:

1. Rational System Definition - An organization is a collectivity oriented to the pursuit of relatively specific goals and exhibiting a relatively highly formalized social structure.
2. Natural System Definition - An organization is a collectivity whose participants are little affected by the formal structure or official goals but who share a common interest in the survival of the system and who engage in collective activities, informally structured, to secure this end.
3. Open System Definition - An organization is a coalition of shifting interest groups which develop goals by negotiation; the structure of the coalition, its activities, and its outcomes are strongly influenced by environmental factors.

Since there are many equivalent ways to organize, style is used to reduce the organizational possibilities in a particular culture. Lammers and Hickson call the cultural styles of organization rational myths. A rational myth typology developed by Lammers and Hickson [47] is as follows:

1. Latin countries tend to have organizations of relatively high centralizations, rigid stratification, sharp inequalities among levels, and conflicts around areas of uncertainty.
2. Anglo-Saxon countries tend to be more decentralized, have less rigid stratification, and more flexible approaches to the applications of rules.
3. Third world countries tend to have parental leadership patterns, implicit rather than explicit rules, and a lack of clear boundaries separating organizational from non-organizational roles.

2.3 Organizations and their Environment

Where an organization ends and its environment begins is not clear. An organization has to exert effort in order to maintain its boundaries. A collection of models and theories deals with different issues in maintaining organizational boundaries and relating the structure and action of the organization to the environment in which it resides. The organization is not a unified entity in reacting to the environment. Different parts of the organization react to different parts of the environment, and different parts react differently to the same parts of the environment. In addition, an organization is

composed of suborganizations, each of which acts as an environment to the other suborganizations.

Buffering strategies are a way that an organization maintains its boundaries with the environment. Buffers separate the interaction of the organizational subcomponents with each other, and the organizational subcomponents with the environment. Scott [59] emphasizes the boundary maintenance function of the buffers; March and Simon [52] emphasize the cognitive limits of the individual within an organization, and how the buffers reduce the number of events which have to be dealt with at once. Galbraith [33] emphasizes the creation of slack resources through the use of the buffers, which reduces the amount of information needed during task execution. The following is Scott's taxonomy of buffering strategies, along with the Ubik mechanism which supports the taxonomic item.

1. **Coding strategies** to reduce the inferencing necessary when a message is received and to increase the power of the knowledge representation. Configurators are the Ubik coded objects. They are discussed in chapter 4.

2. **Stockpiles** such as material inventories and time buffers to reduce the need for real time coordination. An important buffering mechanism within Ubik is the batching of messages. Batching is discussed in chapter 5.

3. **Leveling**, by which the organization attempts to reduce fluctuations in its input or output environments. Leveling entails an active attempt to reach out into the environment to motivate suppliers of inputs or to stimulate demand for outputs. Ubik has a control mechanism called sponsors, which can control the transaction rate by attracting or reducing transactions. Sponsors are discussed in chapter 7.

4. **Forecasting** to anticipate change and recognize patterns and regularities. Ubik continually attempts to monitor its activity in order to create new organizational structures and reorganize existing structures, in order to create a better match between the organizational structure and the organizational action. Monitoring is discussed in chapter 6.

5. **Growth** to control the market. Ubik supports open systems which continually grow and change. Reorganization is discussed in chapter 8.

Typologies, such as buffering strategies, are important in relation to Ubik not because Ubik is built around any particular typology, but because they represent ways which the people who build applications using Ubik can view their applications. Ubik is built so that typologies can be defined within Ubik and operate in a manner expected by the application builder.

Chapter 3

Applications

An organization can be viewed as a toolkit for constructing and processing a range of applications. The toolkit has functions for dealing with the environment, such as sales, purchasing, and accounting. It has functions for maintaining its organization, such as personnel, inventory, and middle management. It has functions which are part of the applications the organization supports, such as design and manufacturing. An organization is continually adapting its structure and action to improve its support of the existing applications and to support new applications. An example of a process which adapts the structure of an organization originally built for one application to another is functional autonomy. Minsky [53] describes this as the idea that specific goals can lead to subgoals of broader character. In the context of organizations, it can be described as a process in which subfunctions of applications become detached from the original application, generalized, and used to support other applications.

Application software systems are organizations of increasing complexity. The early business systems consisted of simple organizations of stand-alone applications. The introduction of real-time, database-oriented applications established an organization out of which grew multiple applications, all sharing the same communications and database facilities. The introduction of distributed workstations is resulting in complex applications which cooperate and compete with each other. Ubik contains facilities to support this coordination and competition. This chapter presents two applications which will illustrate the use of Ubik: a purchasing organization, and a software development organization.

Ubik applications are built out of the basic Ubik computational object called a configurator, as shown in figure 3.1. A configurator consists of the following sections:

- Name - the name of the configurator

- Input - the messages which the configurator is prepared to receive. The receipt of a message will trigger the configurator into action.

- Output - the messages which are sent when the configurator completes processing.
- Link - one or more sections which specify other configurators linked to this configurator. Ubik supports multiple types of links, as discussed in chapter 4 on organizational structure. If a configurator has an unnamed section, this section, by default, is of type link.part.
- Action - the action taken by the configurator. The most common action is associating the input messages to the output messages.
- To - configurators to which this configurator is sent as a message.
- Control - specifies miscellaneous information about how this configurator is used and controlled.

<table>
<tr><td>name</td><td></td></tr>
<tr><td>input</td><td>output</td></tr>
<tr><td colspan="2">link</td></tr>
<tr><td colspan="2">action</td></tr>
<tr><td colspan="2">to</td></tr>
<tr><td colspan="2">control</td></tr>
</table>

Figure 3.1: Ubik configurator format.

3.1 Purchasing Organization

The purchasing organization described here is loosely based on the MIT purchasing system, as specified in the *MIT Guide to Administrative Offices* [5]. The purchase system is a subsystem of a larger organization and shares some of the larger organization's subparts. It is composed of the employees in the role of a purchaser; the purchasing, shipping, and accounting departments; and the concept of an external organization called a vendor. The organization is illustrated in figure 3.2, and the forms used to communicate between the subparts are shown in figure 3.3.

The purchasing system works as follows:

1. The purchaser creates a purchase-requisition and sends it to the purchasing department.

2. The buyers in the purchasing department create purchase-orders from the purchase requisition and send copies of it to vendors and the accounting department. A receiving-ticket is sent to the shipping department, so the employees in that department know how to distribute the purchased item once it arrives.

3. The vendor ships the purchased item to the customer's shipping department. Once a month, on the customer's bill date, an invoice is prepared consisting of all the items ordered by the customer the previous month.

4. The shipping department receives the item, matches it to a receiving ticket, and sends it to the purchaser. It also sends a notification-of-receipt to the purchasing and accounting departments.

5. The accounting department sends the payment to the vendor, once it receives the invoice and notification-of-receipt.

Figure 3.2 is a two-dimensional representation of the purchasing organization. In this example, the purchase-organization configurator has the following sections:

- The name section has the value purchase-organization.

- The link.part section has references to the department and vendor configurators. The department configurator in turn references the purchasing, accounting, and shipping configurators.

- The link.flow section has a description of the message flow between the configurators referenced in the link.part section.

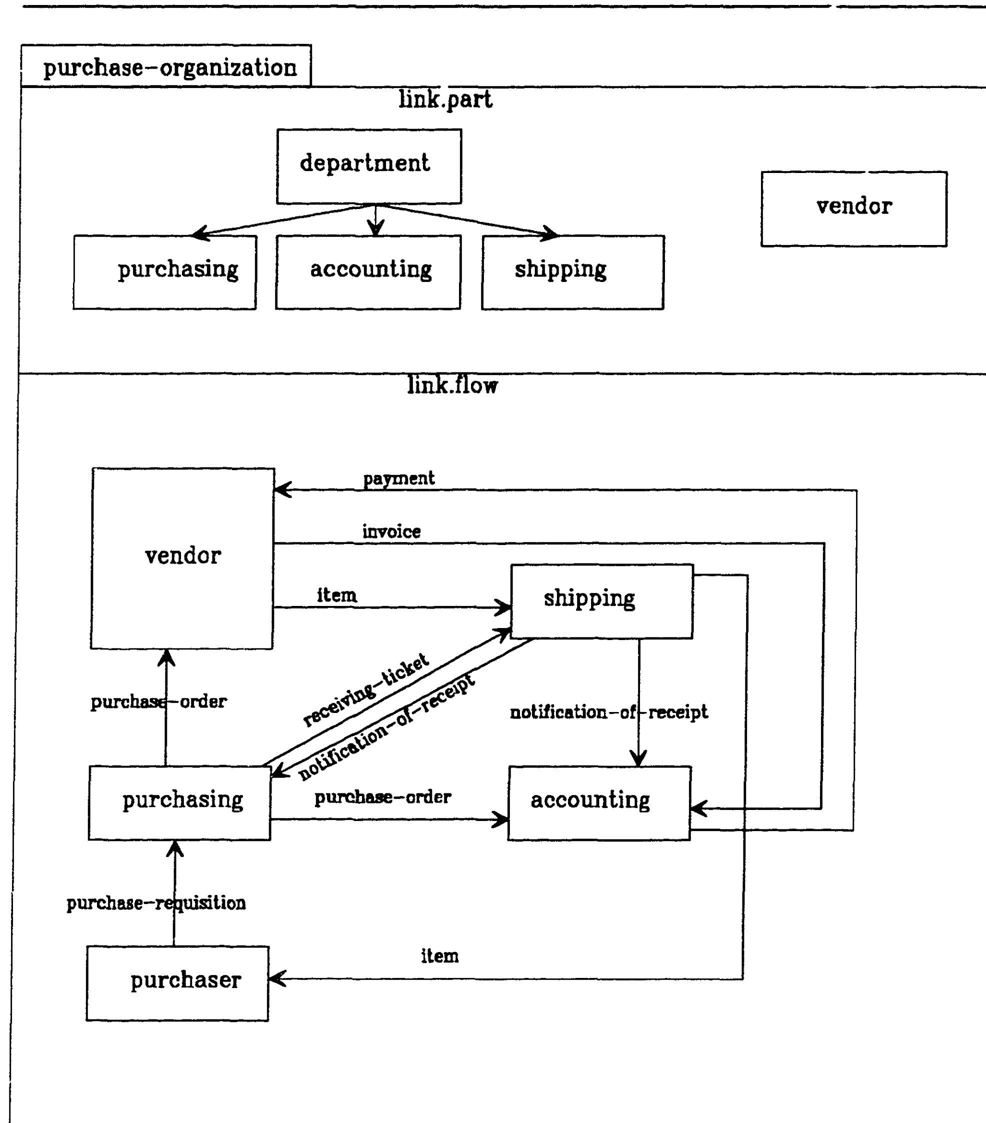


Figure 3.2: The purchase organization is a suborganization used to carry out purchasing activities. The purchase organization consists of three internal departments: purchasing, accounting, and shipping; one external organization: vendor; and employees of the organization who take the role of purchaser. The link.part section specifies the configurators which this configurator references. The link.flow section specifies the message flow between the configurators.

purchase-requisition					
purchaser		po-number		requisition-number	
account-no		date-requested		category	
item					
quantity	description				

purchase-order					
po-number		company		requisition-number	
date-sent		date-requested		category	
vendor		buyer			
item					
quantity	part number	description	unit-price	amount	
			total		

invoice					
po-numbers		company		date	
item					
quantity	part number	description	unit-price	amount	
			total		

packing-slip					
po-number		company		date	
item					
quantity	part number	description			

receiving-ticket					
po-number		purchaser		date	

notification-of-receipt			
po-number		date	

Figure 3.3: Purchase Organization Forms are as follows: purchase-requisition, purchase-order, invoice, packing-slip, receiving-ticket, and notification-of-receipt.

3.1.1 Gradual Automation

Ubik supports the gradual automation of an application. Initially an application, placed into Ubik, closely matches the preexisting non-computerized organization. Gradually, on a localized basis, the application can be automated. Automation will never be complete. Some parts of an application require cooperative interaction with an end-user. The filling out of the purchase-requisition form by an end-user is a cooperative interactive activity between the purchaser and the Ubik system. The initial automation of the filling out of the purchase-requisition form consists of displaying the form on the computer screen. After the purchaser has completed the form, Ubik sends it to the purchasing department. The next stage in automation is for a Ubik action to support an interactive dialogue with the end-user, as illustrated in figure 3.4. In this example the top form shows the fields which the user fills in; the bottom form shows the fields which the system, in an interactive dialog with the user, fills in. When the user fills in a field, the Ubik system immediately fills in all the fields which can be generated from that field.

purchase-requisition					
purchaser	jill	po-number		requisition-number	
account-no		date-requested	3/9/90	category	
		item			
quantity	description				
1	apple laserwriter II				
1	IBM PS/2 80				

purchase-requisition					
purchaser	[illegible]	po-number		requisition-number	235
account-no	56021	date-requested	[illegible]	category	computers
		item			
quantity	description				
1	[illegible]				
1	[illegible]				

Figure 3.4: Purchase-requisition interactive dialog. Ubik displays a purchase-requisition on the computer screen. The user fills in the purchaser, date-requested, and item fields, as shown in the top form. Ubik fills in the account-no, requisition-number, and category, as shown in the bottom form.

3.1.2 End-user Programming

End-user programming is facilitated in Ubik by allowing the user to program within two-dimensional pictures of the objects. An end-user program to support the purchase-requisition interactive dialogue is shown in figure 3.5. A variable in Ubik is a symbol preceded by a question mark. Variables are used to link the various fields in the form with the configurators which specify the action taken.

The configurator with the name `interactive-purchase-requisition` represents the program. The configurator has no input section. The output section specifies that after the configurator is evaluated, the purchase-requisition is sent to the purchasing department. The action section consists of a collection of configurators which perform as follows:

1. The unnamed configurator's control section has a display command which will display the purchase-requisition on the end-user's display device. The fields are filled in the following order:

   (a) The requisition-number is filled when the form is displayed. This is indicated by variable ?R which links to the pr-no configurator.

   (b) When the user fills in the purchaser field, the account-no configurator triggers, filling in the account-no field.

   (c) When the user fills in the description field, the category configurator is triggered, filling in the category field.

2. The `account-no` configurator fills in the account-no on the purchase-requisition form. It is triggered when `?X` receives a value by the user filling in the purchaser field, and it produces a value for `?Y`. It consists of the configurator `account-table`. The table associates the `?X` with the `?Y` variable.

3. The `pr-no` configurator has an output section and no input section. The configurator is automatically triggered to produce a value for the `?R` variable, which is the requisition-number. The value will appear in the purchase-requisition form without any user interaction. The action is not shown, but can consist of a Ubik program or a program in any language whose interface is supported by Ubik. The action will produce the next requisition-number.

4. The `category` configurator finds the category from the description. Its action is not shown. The action can consist of a simple table lookup as in the `account-no` box, or a complex program to attempt to find the category from an unformatted text description given by the user. The action might consist of a further dialog with the end-user.

After the purchase-requisition form is filled out, it is sent to the purchasing department, as shown in figure 3.6.

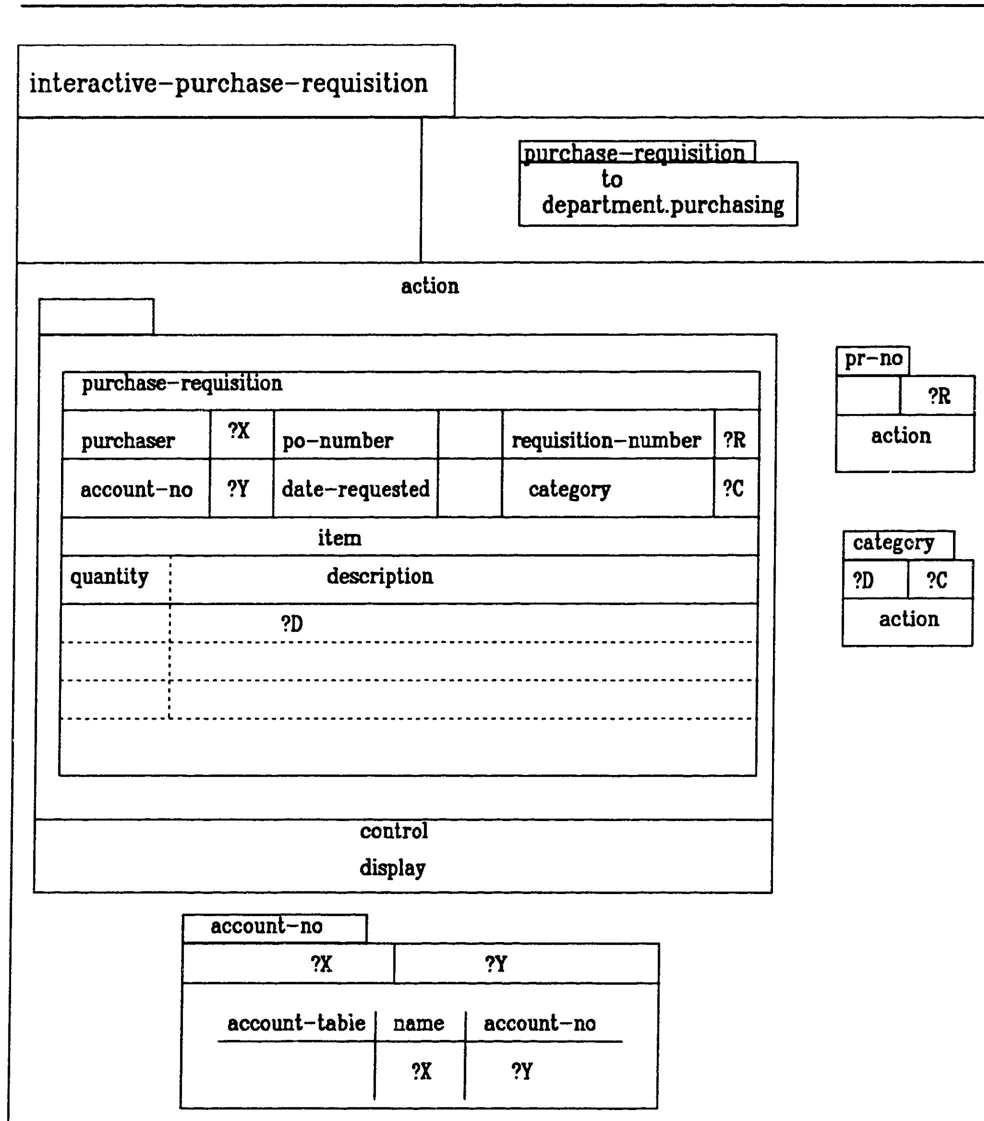


Figure 3.5: End-user written purchase requisition dialog.

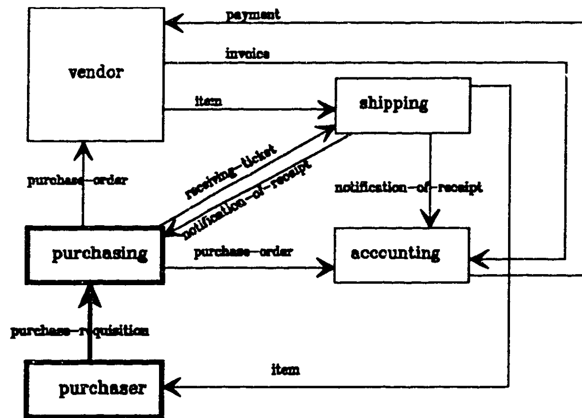


Figure 3.6: Flow of purchase-requisition form to purchasing department.

3.1.3 Bureaucracy

An organization is represented by a collection of linked configurators. Since each configurator within an organization can take organizational action and direct organizational messages, the configurators can be said to form a bureaucracy. An example of an organization with bureaucratic decisions is shown in figure 3.7. This example operates as follows:

1. The message is received by the `department.purchasing` configurator. It is forwarded to the `sections` configurator, as specified by the `flow-.purchase-requisition` link.

2. The message is received by the `sections` configurator. The input section of this configurator contains a purchase-requisition configurator. The purchase-requisition configurator has a `link.part` section and a `to` section. The `link.part` section specifies that the variable `?cat` is to be bound to the value of the category field on the form. The `to` section specifies that the purchase-requisition is to be sent as a message to a configurator, with the name specified in the `?cat` variable.

3. The message is received by one of the configurators named in the category field. This configurator will forward the message to its buyer configurator, as specified by the flow link.

4. The message is received by a `buyer` configurator. The buyer produces purchase-orders from the purchase-requisition, as shown in figure 3.8. In figure 3.8 there are two items in the purchase-requisition, each of which is processed by a different vendor. Figure 3.9 shows the purchase-order being transmitted to the vendor.

   A more complex bureaucratic decision is shown in section 5.4.

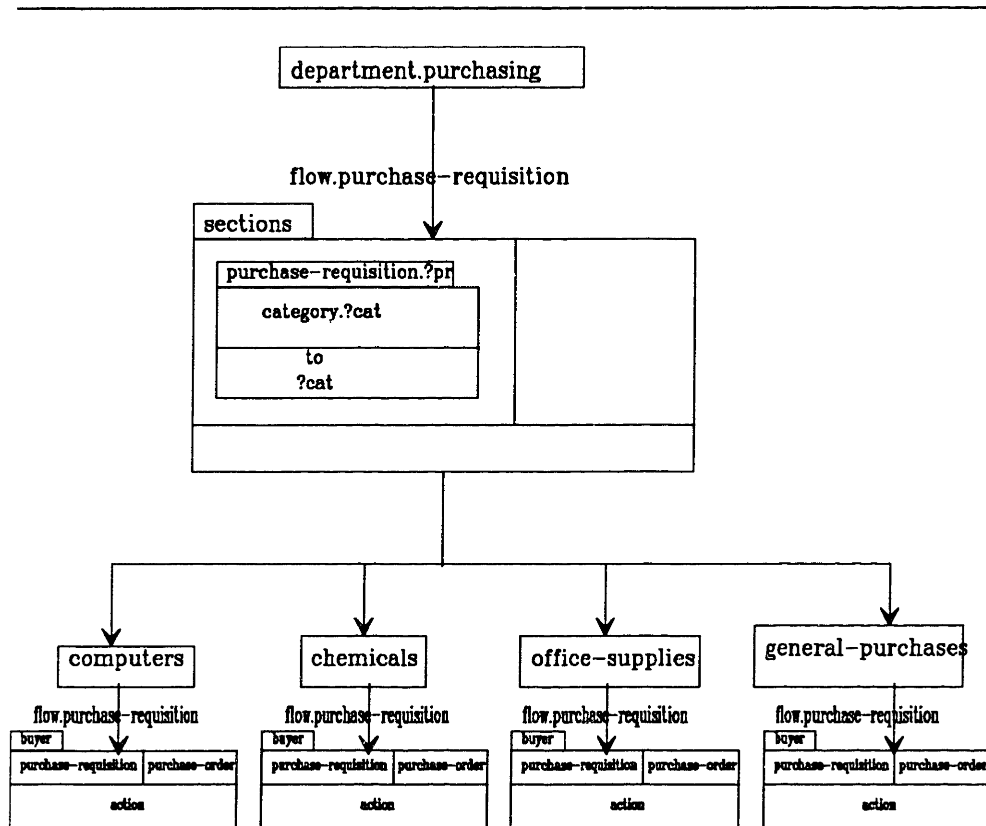


Figure 3.7: Bureaucratic decision making in the purchasing department at the **sections** configurator. This configurator determines which buying section will receive the purchase-requisition.

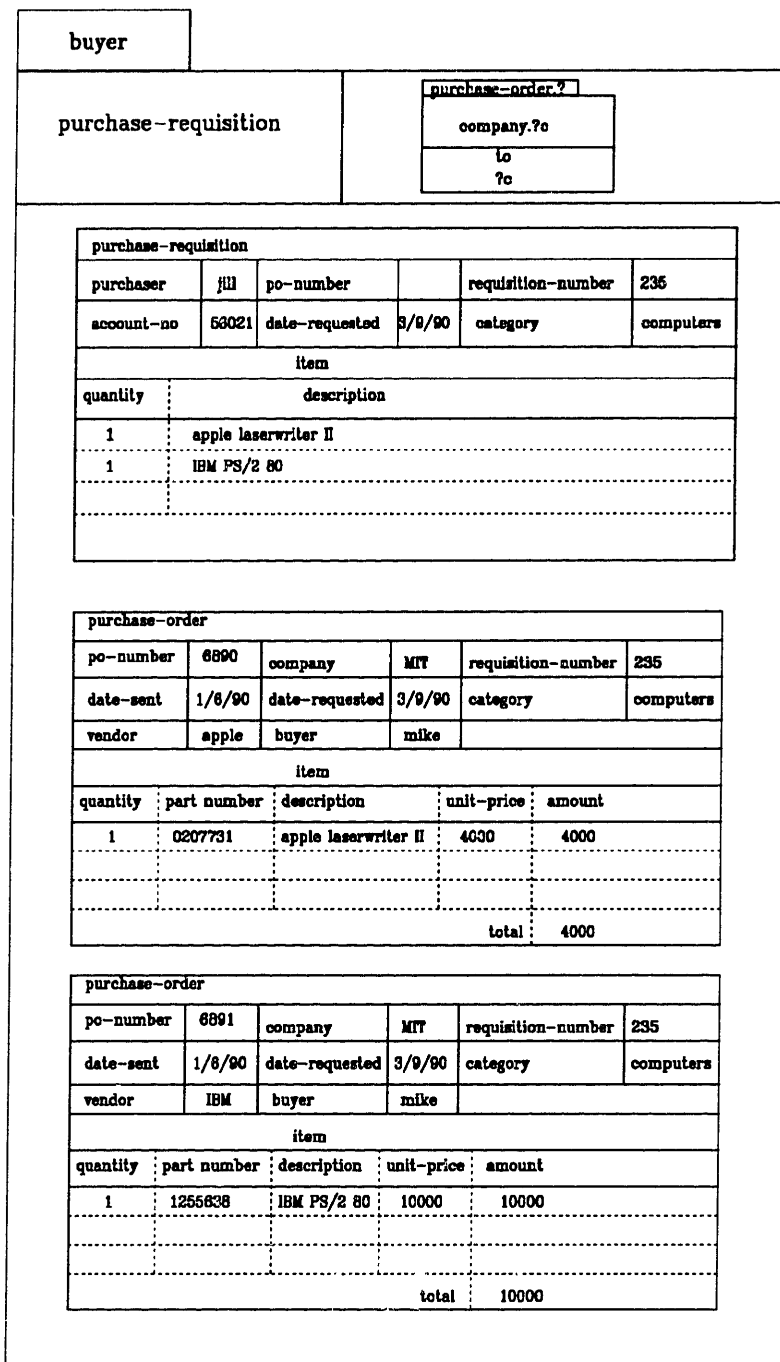


Figure 3.8: Buyer creating purchase-orders from the purchase-requisition.

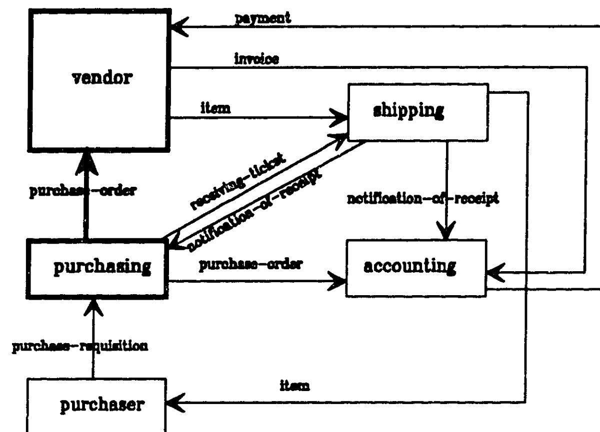


Figure 3.9: Flow of purchase-order from purchasing department to vendor.

3.1.4 Batching

Batching is the accumulation of messages at the input section of a configurator. Batching is specified with an *and expression* of input message patterns in the input section of a configurator. The configurator is not triggered until all the messages in the *and expression* arrive.

The shipping department in the purchasing application receives a receiving-ticket and notification-of-receipt forms. The receiving-ticket is sent by the purchasing department. The notification-of-receipt is created from the packing slip of an item received by the shipping department. For a particular purchase-order, both the receiving-ticket and notification-of-receipt must be matched before the item received can be sent to the purchaser. In figure 3.10, a collection of receiving-tickets and notification-of-receipts are batched on the input section of the shipping department. The forms are matched for purchase-order 6860 and 6890. These forms can be processed by the shipping department. The receiving-ticket for purchase-order 6891 cannot yet be processed.

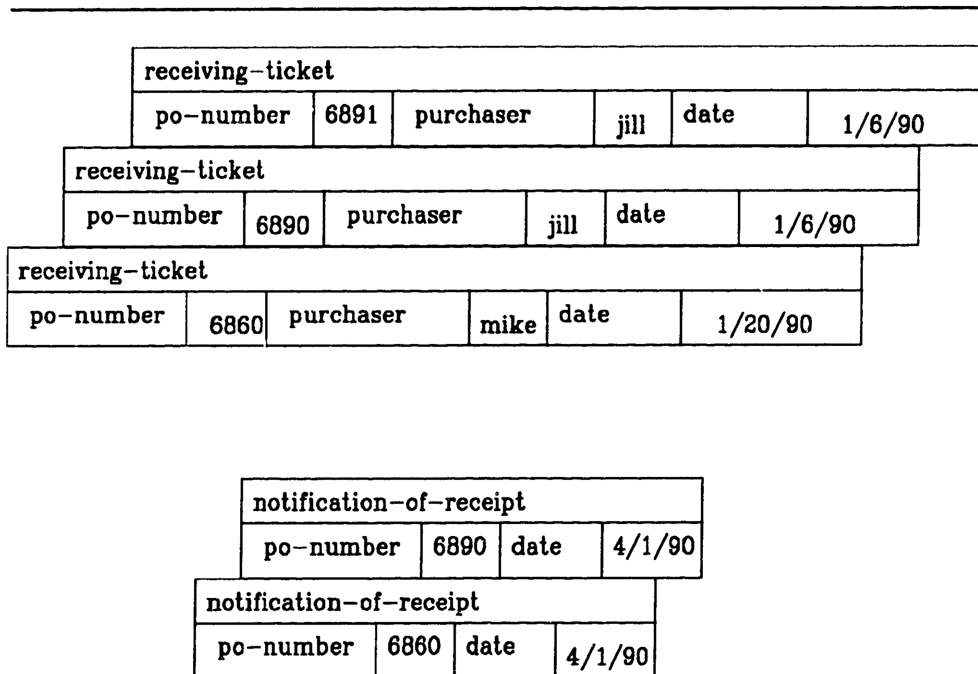


Figure 3.10: Batching of shipping department forms.

The end-user program for the batching at the shipping department is shown in figure 3.11. The input section *and expression* has two operands: receiving-ticket and notification-of-receipt. They are linked by the ?PO variable on the po-number field. For the configurator to trigger, the two forms have to be received with matching po-number fields. After triggering, the

output section in figure 3.11 specifies that the notification-of-receipt is distributed to the accounting and purchasing departments, as shown in figure 3.12.

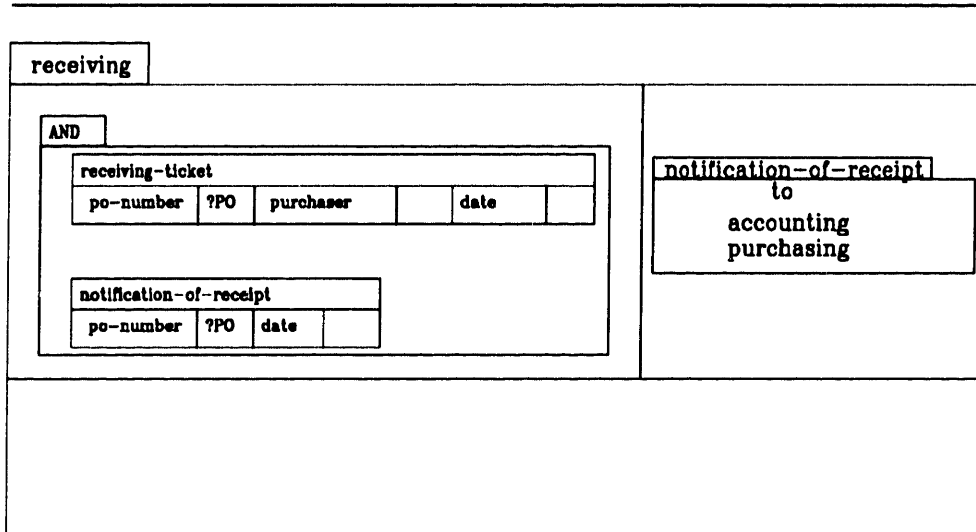


Figure 3.11: End-user written program to batch forms.

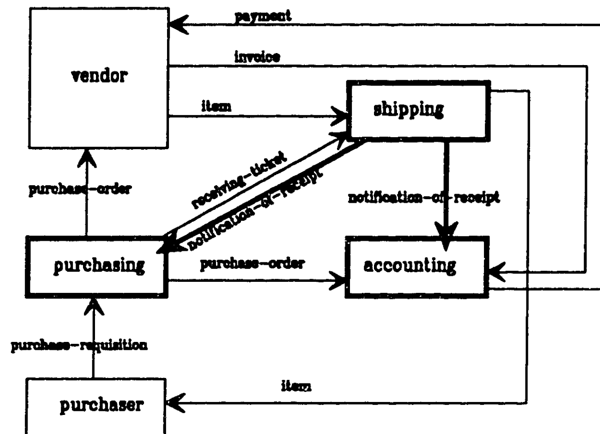


Figure 3.12: Flow of notification-of-receipt form to accounting and purchasing departments.

Another example of batching is the accumulation of purchase-orders to produce an invoice. The invoices are produced once a month on the customer's bill date, as shown in figure 3.13. In this example, three events need to occur before the `billing` configurator will trigger:

1. A purchase order for the company has to have arrived. The variable

?po is bound to the purchase-order; the variable `?com` is bound to the company issuing the purchase-order.

2. The time of day is 16:00, as specified in the control section.

3. The bill day for the customer is today. The action section of the purchase-order contains an *and expression* which references a quester and a boolean expression. A quester is a configurator which sends itself through the organization seeking information. In this case the quester sends itself to the customer-file configurator to find the bill day for the company. The quester has an input section which specifies the company in variable `?com`, and returns the bill date in variable `?bill-day`. The boolean expression matches the bill day with today's date. If they match, the billing configurator will process all the batched purchase-orders for the company and produce an invoice.

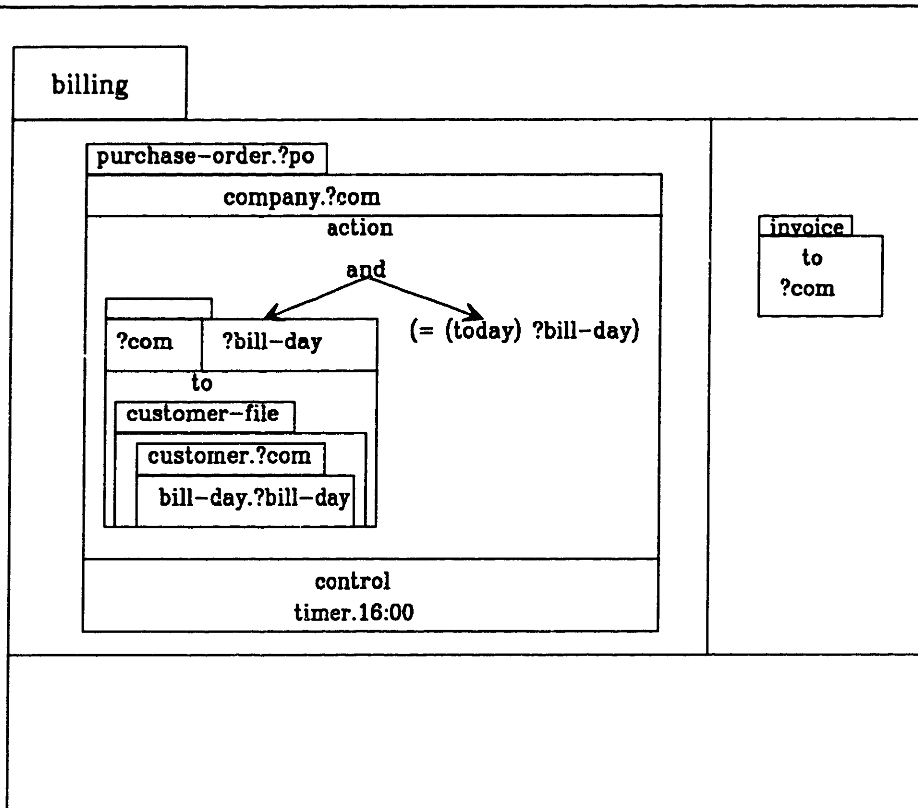


Figure 3.13: Purchase-orders are batched at the billing configurator until the following three events occur: a purchase-order arrives, the purchasing company's bill date is today, and the time is 16:00.

3.1.5 Regrouping

Regrouping is a remapping process in which parts of multiple input messages are mapped onto a new message. The production of an invoice requires the remapping of information from multiple purchase-orders. In figure 3.14, invoice 367334 is produced from purchase-orders 6860 and 6891. The invoice po-number, and item fields contain accumulated information.

The end-user program for automatically producing the invoice from the purchase-orders is shown in figure 3.15. The program works as follows:

1. The invoice po-number field contains a (union ?po) operation. This operation accumulates the purchase-order numbers from all the purchase-orders. The `?COM` variable restricts the accumulation to purchase-orders from the same company.

2. The item fields accumulated information from all the purchase-orders specified by the union operation and company restriction. For each part number, description, and unit-price a new line item is created. The `(+ ?Q)` expression sums the quantity for each part number from all the purchase orders. The `(* quantity.? ?U)` expression multiplies the quantity for each line item by the unit price. `quantity.?` references the contents of the invoice quantity field for this line item. The `(+ amount.?)` expression sums all the amount fields on the invoice form to create the total amount. The `amount.?` expression references all the amount fields on the invoice.

3. The `invoice-no` configurator produces a new invoice number.

4. The `date` configurator produces the current date.

After the invoice is produced, the output section of figure 3.15 specifies that the invoice is sent to the company which issued the purchase-orders, as shown in figure 3.16.

billing

purchase-order.?po
company.?com
action
and
?com ?bill-day
to
customer-file
customer.?com
bill-day.?bill-day
(= (today) ?bill-day)
control
timer.18.00

invoice
to
?com

purchase-order					
po-number	6891	company	MIT	requisition-number	236
date-sent	1/6/90	date-requested	3/9/90	category	computers
vendor	IBM	buyer	mike		

item				
quantity	part number	description	unit-price	amount
1	1255635	IBM PS/2 80	10000	10000
			total	10000

purchase-order					
po-number	6860	company	MIT	requisition-number	583
date-sent	1/20/90	date-requested	3/10/90	category	computers
vendor	IBM	buyer	mike		

item				
quantity	part number	description	unit-price	amount
1	1255638	IBM PS/2 80	10000	10000
			total	10000

invoice					367334
po-numbers	6860 6891	company	MIT	date	3/30/90

item				
quantity	part number	description	unit-price	amount
2	1255638	IBM PS/2 80	10000	20000
			total	20000

Figure 3.14: Creating an invoice from multiple purchase-orders.

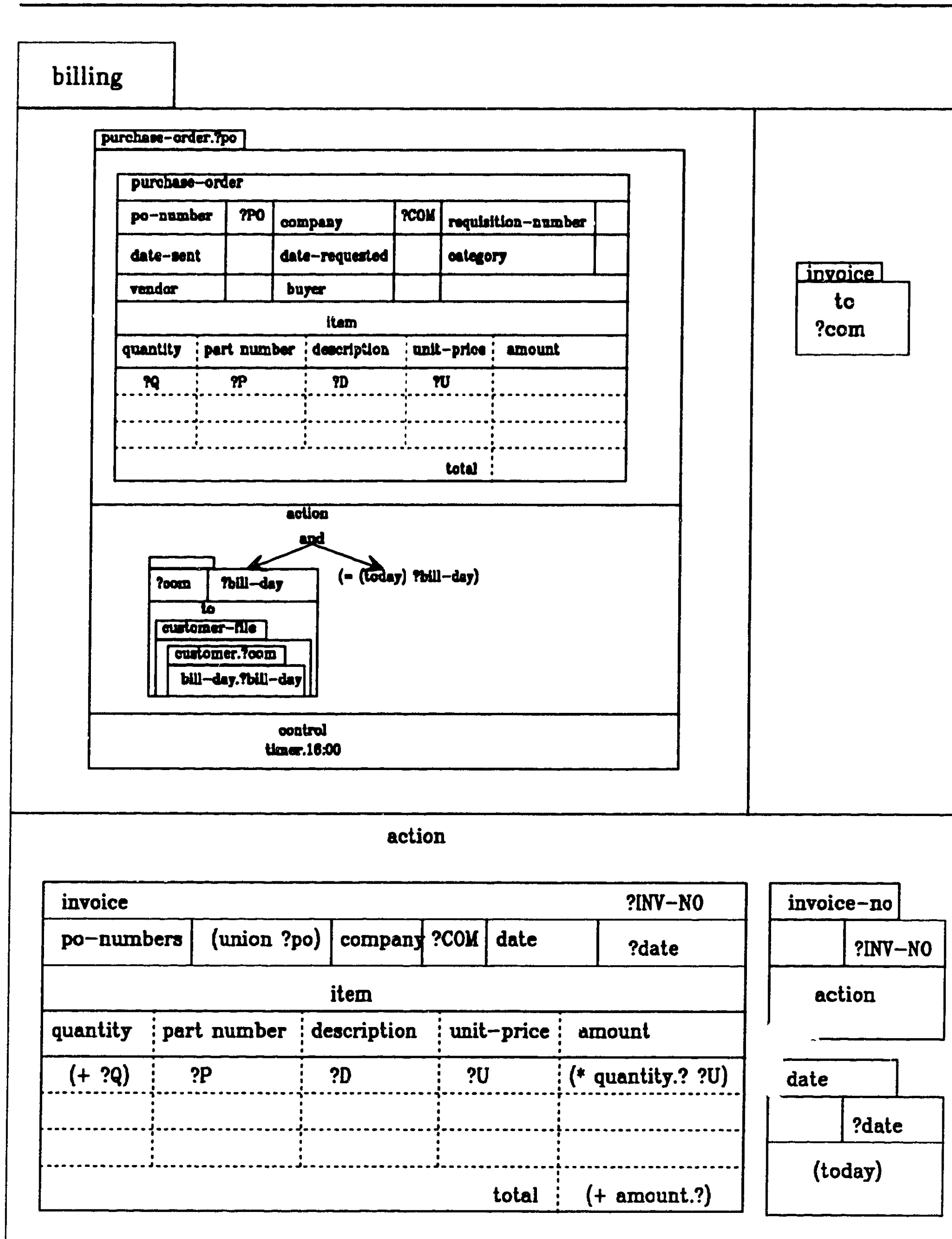


Figure 3.15: End-user written program to create an invoice from multiple purchase-orders.

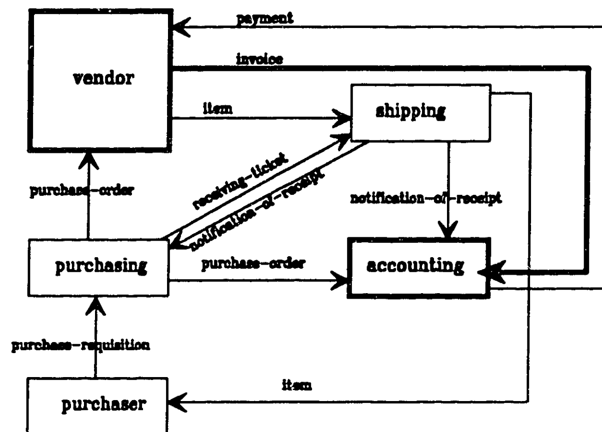


Figure 3.16: Flow of invoice from vendor to accounting department.

3.1.6 Questers

Questers are configurators which search the organization. The input section contains input messages to the quester; the output section contains the message which gives the result of the search. The `to` section contains the configurators which are being searched.

In figure 3.17, a quester is searching for the purchasers who are ordering products from a specified vendor. The input section contains the vendor, and the output section contains the resulting purchasers. There are two forms which must be searched to find the results. The purchase-requisition form contains the purchaser; the purchase-order form contains the vendor. The forms are linked on the requisition-number field.

In figure 3.18, a quester is searching for all departments which accept a purchaser-order as input. The query is on the name of the departments whose input section contains the message purchase-order. *? is a wild-card which will transitively follow links from the link.part section. The wild-card (*? ?dept) will transitively search all the part links in the purchasing organization. The wild-card (*? purchase-orders) will transitively search all part links in the input section for a purchase-order configurator. The result will be bound to the `?dept` variable.

Questers are used for both open and closed searches. In figure 3.17, two files are searched. Since these files can be explicitly located, the quester can answer queries to these files accurately. This would be a closed search. In figure 3.18 the organization as a whole is searched. Since the organization can not be frozen while the search is being conducted, the accuracy of the query depends on the stability of the organization. If an organization's structure and contents are changing during the search, the query can only return an approximate result. A quester seeking information throughout an organization would have to be transmitted to many distributed locations. The accuracy of the results depends on the amount of resources which are given the quester and how long one is willing to wait for the answer. This is an open search.

quester.purchasers

?COM	?PUR

to

file.purchase-requisition

purchase-requisition

purchaser	?PUR	po-number		requisition-number	?PR
account-no		date-requested		category	

item

quantity	description

file.purchase-order

purchase-order

po-number		company		requisition-number	?PR
date-sent		date-requested		category	
vendor	?COM	buyer			

item

quantity	part number	description	unit-price	amount
			total	

Figure 3.17: Quester to find purchasers of a specified vendor's product.

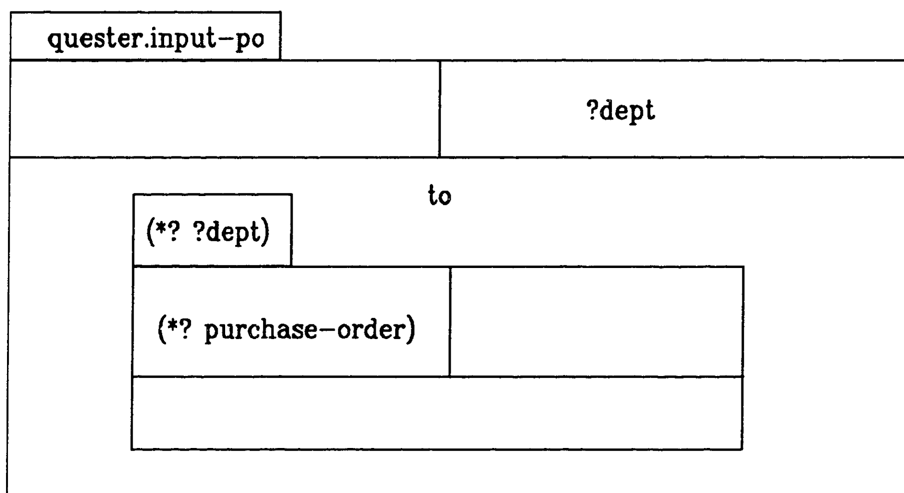


Figure 3.18: Questers to find departments which accept purchase-orders.

3.1.7 Parasitic Tapeworm

Tapeworms are configurators which can be attached to other configurators. They are used as monitors or censors. A configurator is a tapeworm if the tapeworm command appears in its control section. A tapeworm attaches itself to the configurators which are specified in its input section. In figure 3.19 a monitor tapeworm is attached to a purchase-order. This tapeworm will trigger on the due date of the purchase-order, as specified in the date-requested field. When the tapeworm triggers, it will send a copy of the purchase-order to the buyer. The tapeworm will travel with the purchase-order as the purchase-order is sent throughout the organization. This tapeworm is part of an expediting subsystem which tracks and facilitates purchases. A tapeworm used in this manner is called a parasitic tapeworm.

To insure that a parasitic tapeworm is placed on all purchase-orders, another tapeworm is attached to the buyers in the purchasing department, as shown in figure 3.20. This tapeworm attaches the parasitic tapeworm to a purchase-order when the purchase-order is created and transmitted by a buyer. This tapeworm triggers on the sending of a purchase-order, as specified in the control section by the tapeworm command. The tapeworm command specifies that this is a monitor which triggers on a send operation. When this tapeworm triggers, its action part installs the overdue-purchase-order-tapeworm on the purchase-order. Section 6.4.2 gives a more detailed description of parasitic tapeworms.

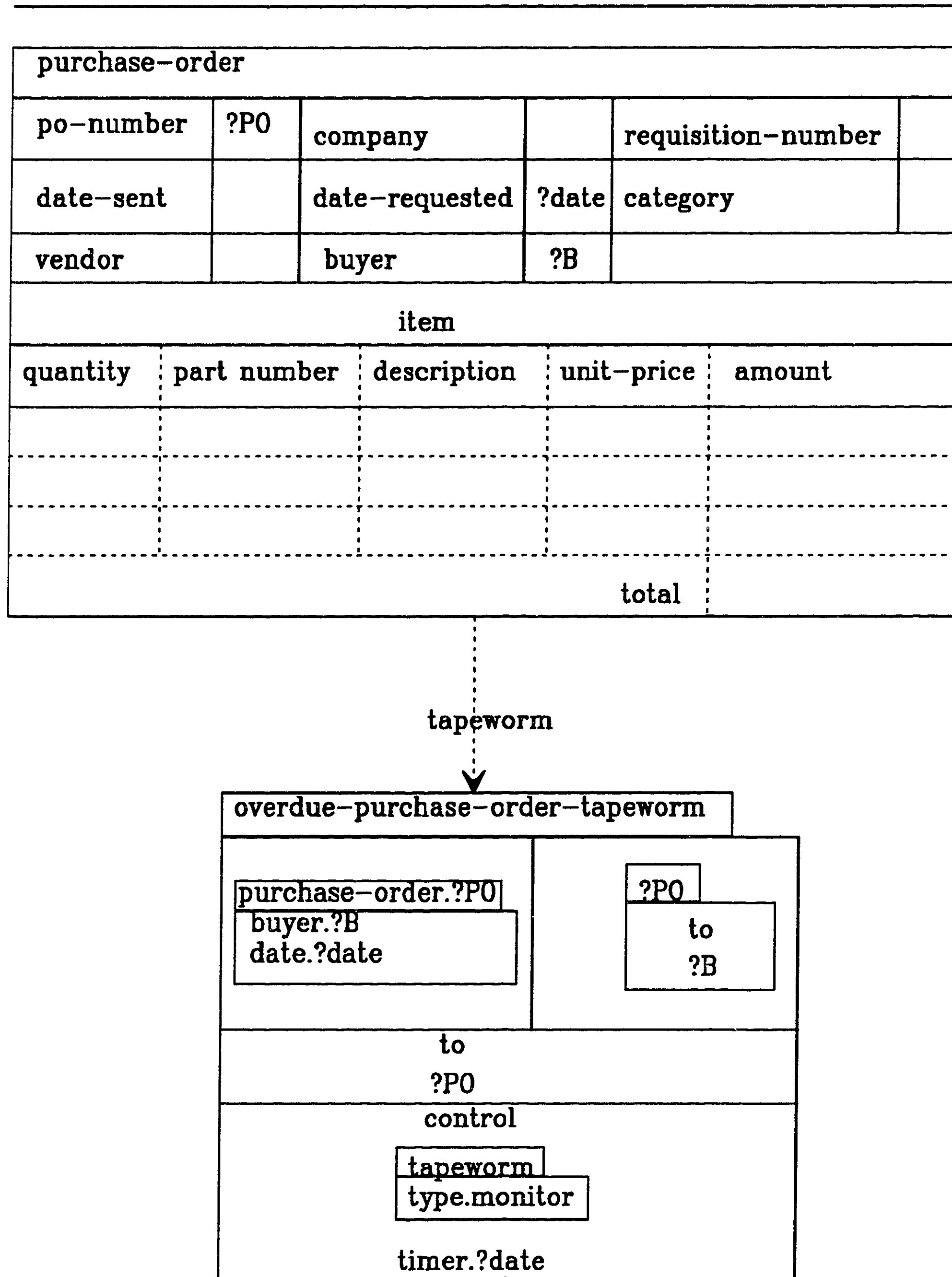


Figure 3.19: Parasitic tapeworm on purchase-order which triggers when purchase-order is overdue.

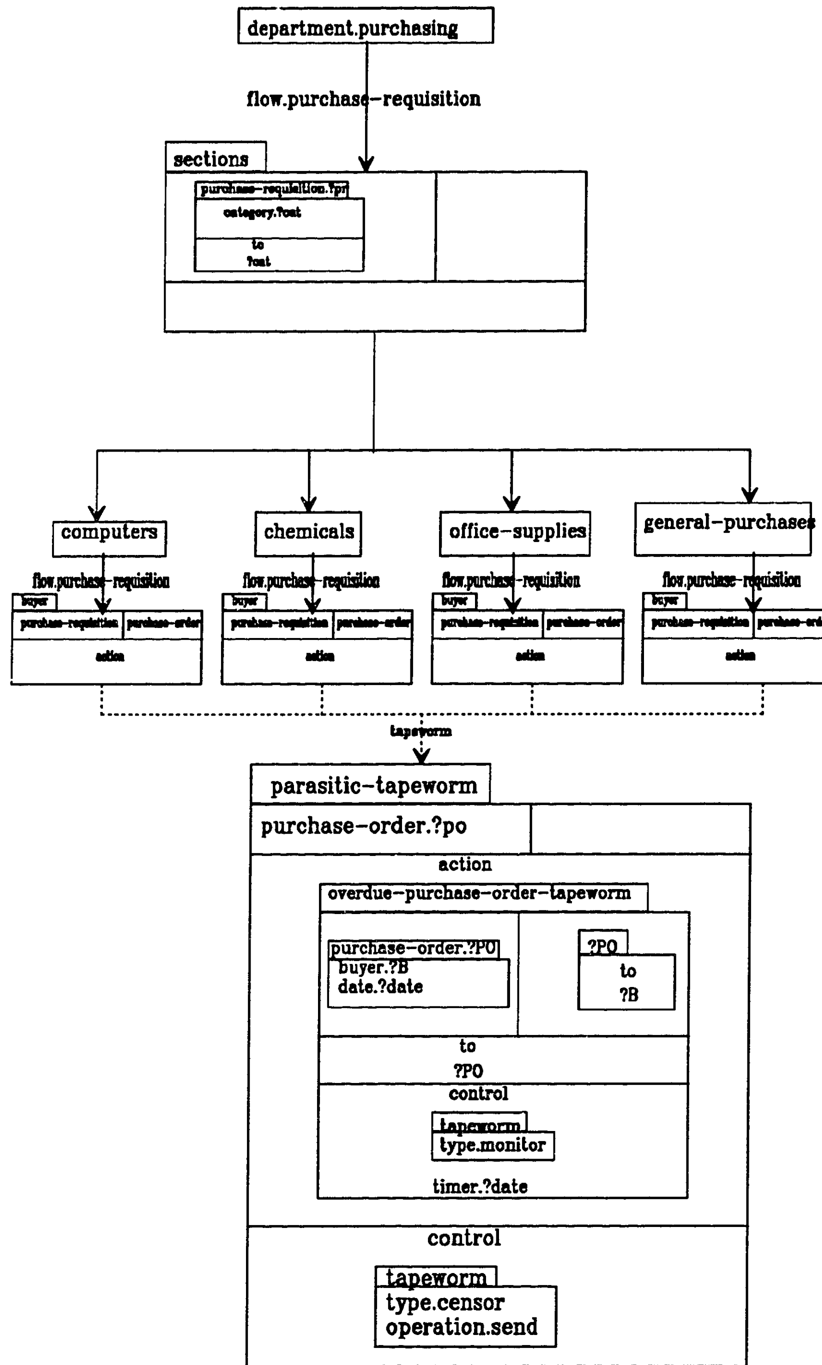


Figure 3.20: Installation of parasitic tapeworm in purchasing department.

3.1.8 Freedom of Action Tapeworm

A freedom of action tapeworm is the use of a tapeworm to monitor the action of other configurators, and to report when the action exceeds specified limits. A manager will use a tapeworm in this manner to monitor the work of his employees. Figure 3.21 shows a freedom of action monitor which triggers when a buyer sends a purchase-order. It sends a purchase-order to the buyer's manager if the buyer exceeds the maximum item cost on a purchased item. This tapeworm's action section contains an **and** expression with three references, as follows:

1. A purchase-order form which has a purchase-order number `?PO` as input and finds the part number `?PN` and unit-price `?PU`.
2. A maximum-item-cost table which looks up for the part-number `?PN`, the max-cost `?MC`.
3. A boolean expression which finds if the unit-price is greater than the max-cost. If it is, the tapeworm's output message is sent.

The freedom of action tapeworm is installed such that it monitors all purchase-order output of the buyers, as shown in figure 3.22. Section 6.4.1 gives a more detailed description of freedom-of-action tapeworms.

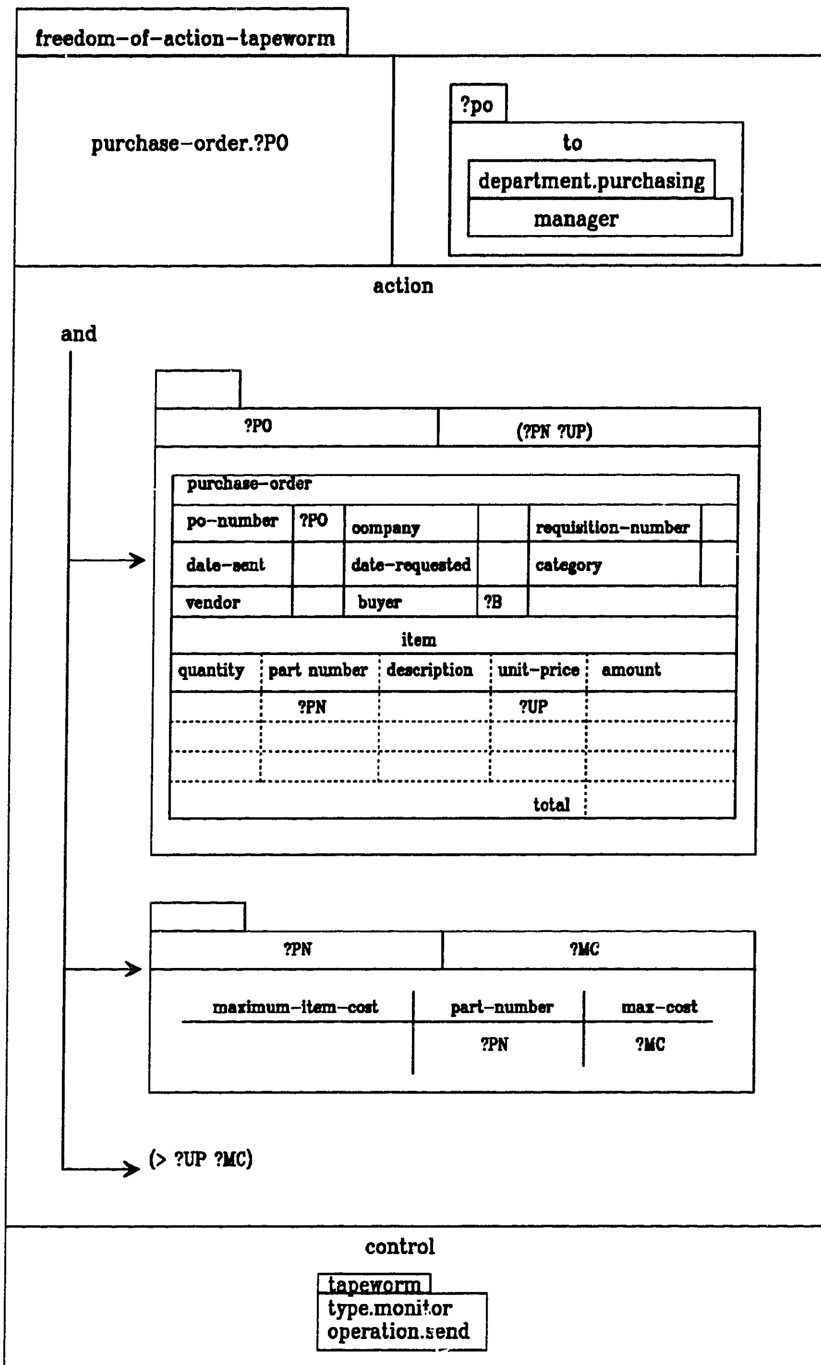


Figure 3.21: Freedom of action tapeworm, in which a manager monitors the buyers' pricing of purchased items.

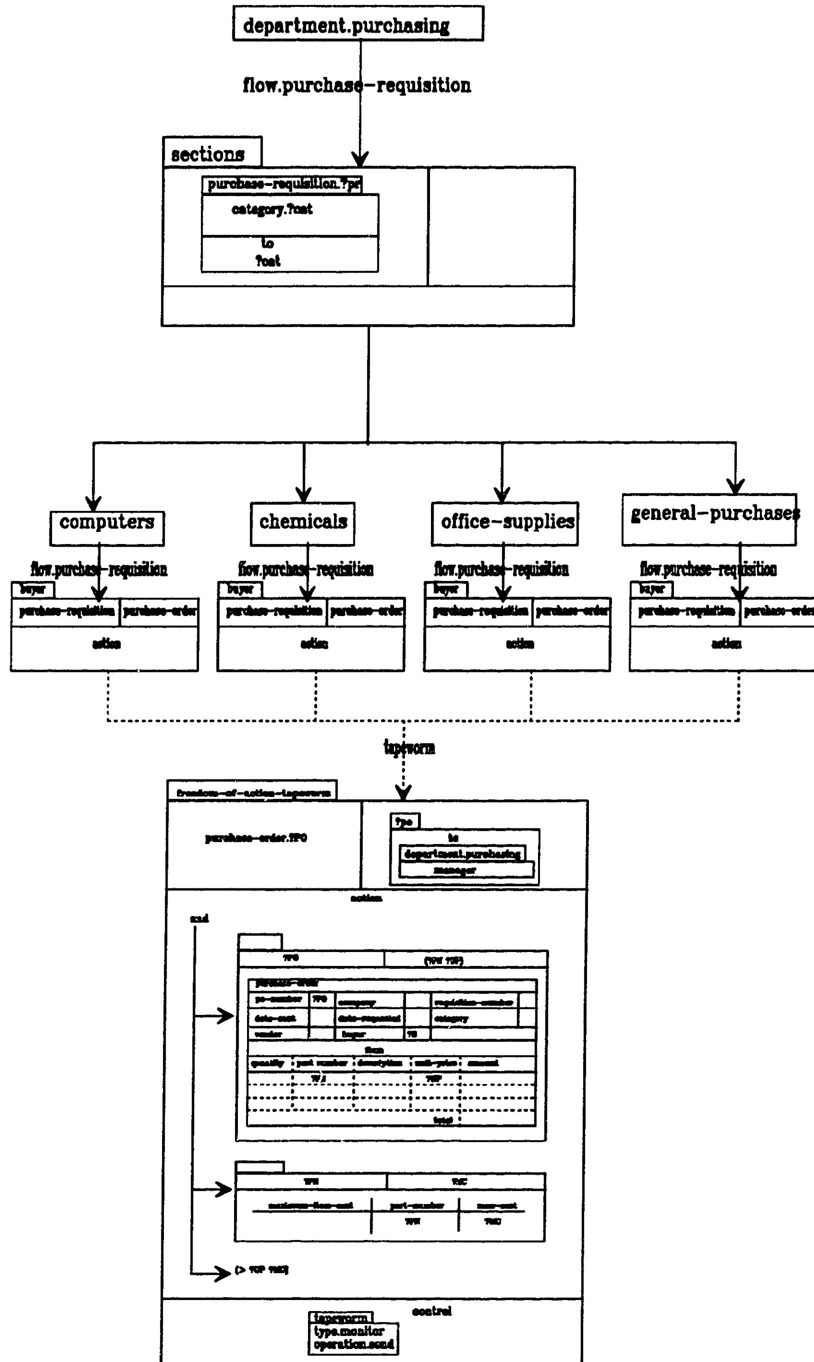


Figure 3.22: Installation of freedom of action tapeworm in purchasing department.

3.2 Development of Large Software Systems

One important application area of Ubik is the development and maintenance of large software systems. The model of software development described in this section requires four sub-organizations: version dependency, functional dependency, project management, and organizational flow. Each sub-organization is separately defined. Message flow is used to coordinate the actions of all four suborganizations.

3.2.1 Version Dependency Sub-organization

A software system consists of multiple programs, each of which has multiple versions. A software system also consists of subsystems, each of which also has multiple versions. The version dependency sub-organization coordinates the collecting of programs into subsystems, and subsystems into released systems. This sub-organization is based on the model supported in the System Building Systems [26]. Figure 3.23 shows a version dependency sub-organization.

Version dependency performs two types of control:

1. Program versions - two taxonomies, versions and history, keep track of the versions for a program. The version taxonomy contains the programs and the versions for each program. The history taxonomy contains the programs and the relationship between each version for a program. For example, program A contains three versions: 1,2 and 3, Version 2 and version 3 were both created by modifying version 1.

2. Subsystem versions - the department.testing maintains four subsystems: subsystem-X, subsystem-Y, integration, and released-system. The subsystems are related to each other, in that subsystem-X and subsystem-Y are combined to become the integration systems. The integration system eventually becomes the released system. A process called promotion moves the programs in a subsystem into the next subsystem. Promotion is an on-going process, in that new systems are always being developed and released. Each subsystem represents a testing level. There are criteria developed by the testing department which determine when a system is ready for promotion.

A coordination problem occurs when fixes are made to subsystems which will be promoted into. For example, if a fix is made to the released system to quickly correct a problem which users of the system have encountered, and the corresponding fix is not made to the integration subsystem, subsystem-X, and subsystem-Y, then when promotion replaces the released system, the error will reappear. The version dependency sub-organization prevents this problem from occurring by keeping careful track of the version dependencies. In this example, when subsystem-X is promoted into integration, the version

dependency sub-organization will discover that program A version 3 is not a derivative of program A version 2. They are both derivatives of version 1. The functions represented by version 2 and version 3 will have to be combined to assure that no fixes are lost.

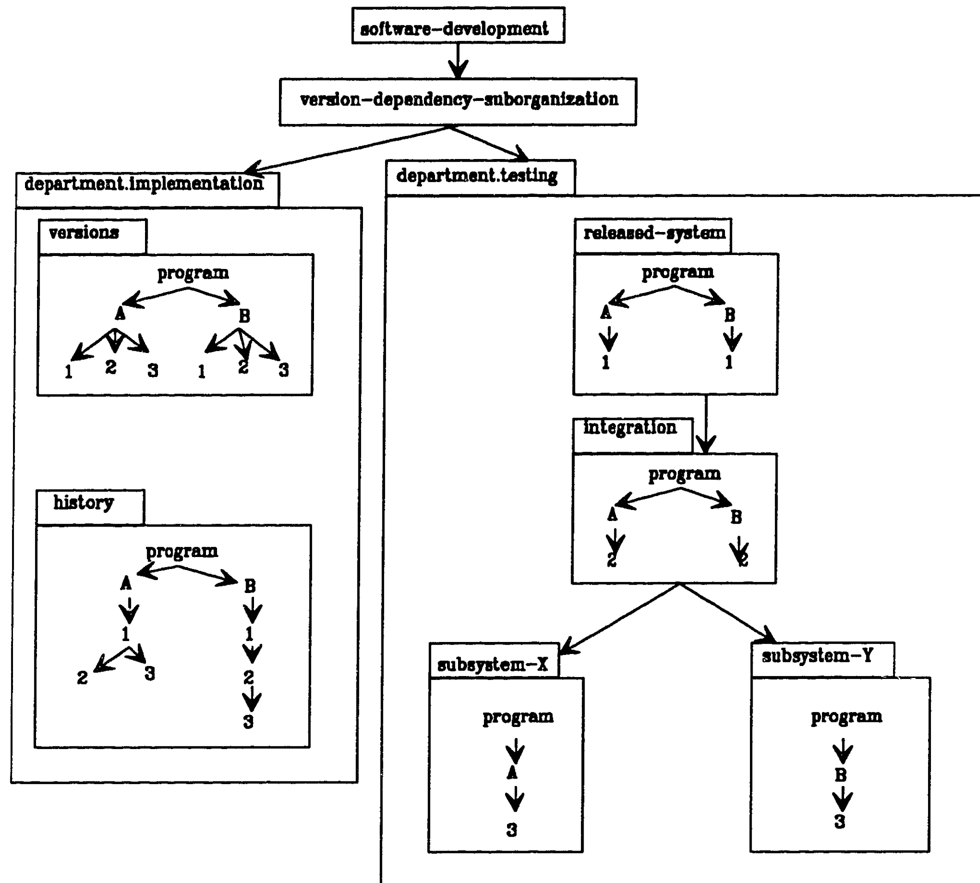


Figure 3.23: Version dependency sub-organization

3.2.2 Functional Management Sub-organization

Software systems can be specified in terms of the functions which comprise the system. The functional composition of the system does not necessarily correspond to the program composition. For example, multiple programs might have to be changed to support a new device, such as CD ROM. These programs already support other functions, so that the mapping of functions to programs is many-to-one, and the mapping of programs to functions is also many-to-one. Figure 3.24 shows a functional management sub-organization.

A program is changed for a reason. The reason is represented by a PTM, program trouble memo, or a DM, design memo. These memos specify the

function which the change supports. The function taxonomy contains, for each function, the programs which support the function, and the memos which describe the function.

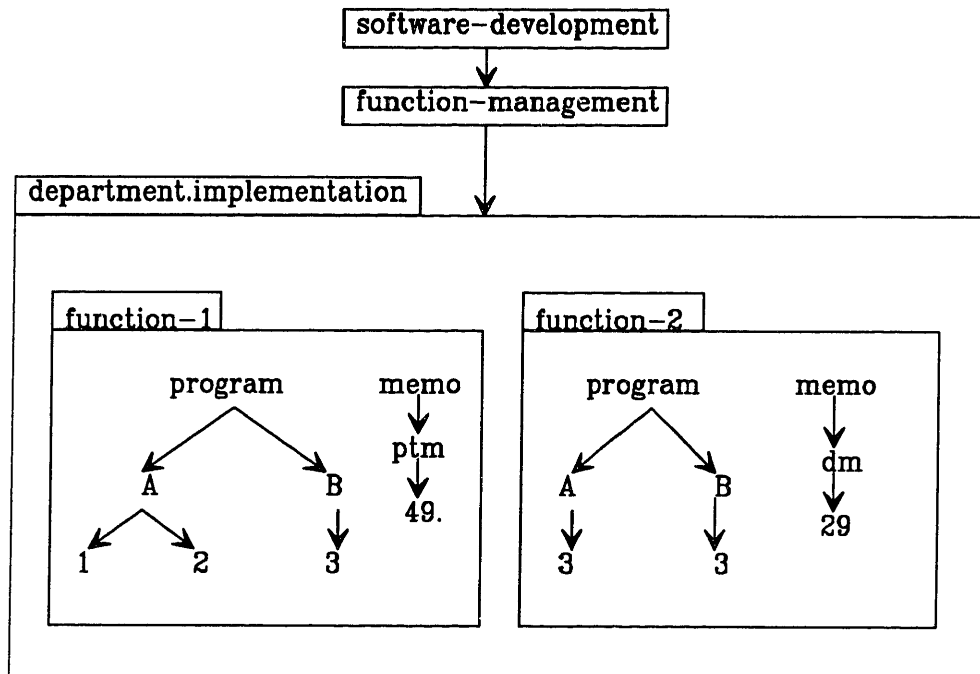


Figure 3.24: Function management sub-organization

3.2.3 Project Management Sub-organization

Software systems are constructed by programmers over a period of time. The project management sub-organization contains the resources required to implement a software system. A project consists of people, implementation phases, schedules, collections of programs, and locations, some of which are shown in figure 3.25. In this example only the flow links are shown. There are three projects. Each project consists of the phases and the program flow between the phases. Project-1 and project-2 consist of phases implement and component-test. The programs flow from implement to program test. Project-3 consists of the phase integration test. It receives programs from project-1 and project-2.

3.2.4 Organizational Model

The organizational model relates the various organizational departments which are involved in the specifying and building of the software system.

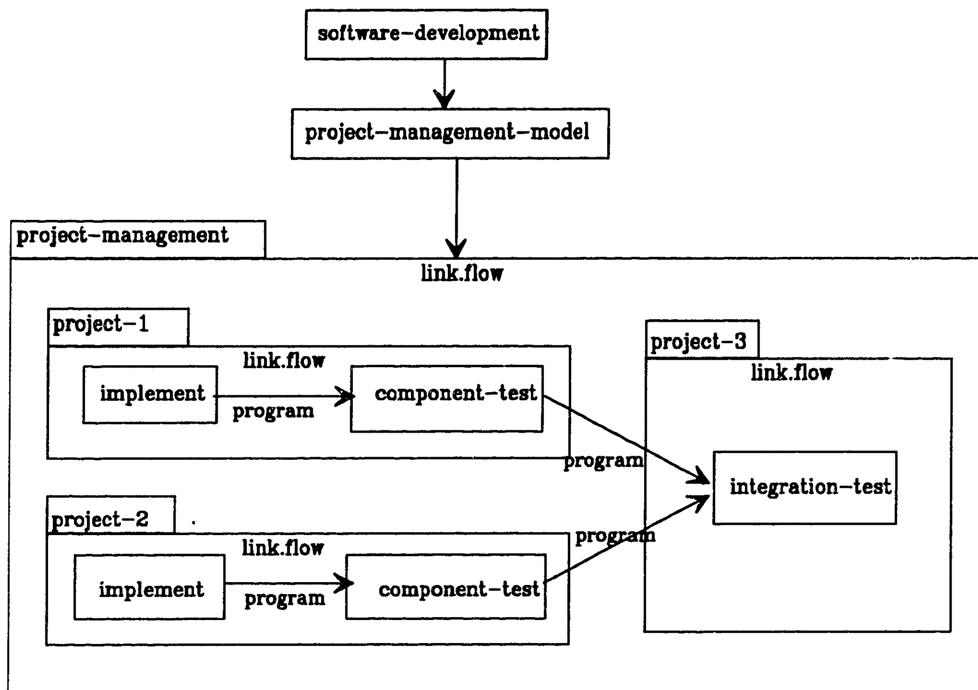


Figure 3.25: Project management sub-organization

The software project, as shown in figure 3.26, involves the following departments: design, programming, documentation, test, and release. The actions of the various departments are coordinated with messages as follows:

1. DM - the design memo is sent from the design department to the implementation and documentation departments. The implementation department constructs programs according to the design. The documentation department produces manuals based on the design.

2. Manual - is produced by the documentation department and sent to the design, implementation, and testing departments. It is also sent to the customers.

3. PTM - the program trouble memo is created by and sent from the department which finds the problem. It is received by the release department from customers who encounter a problem. The PTM is sent to the implementation department, where a fix is made. The implementation department sends it to the design department, where it can become the basis for a design change.

4. Programs - flow from the implementation department to the testing department, and from the testing department to the release department, from which they are sent to the customers.

5. Promotion - is a message which initiates promotion action. Promotion initiates the movement programs between subsystems.

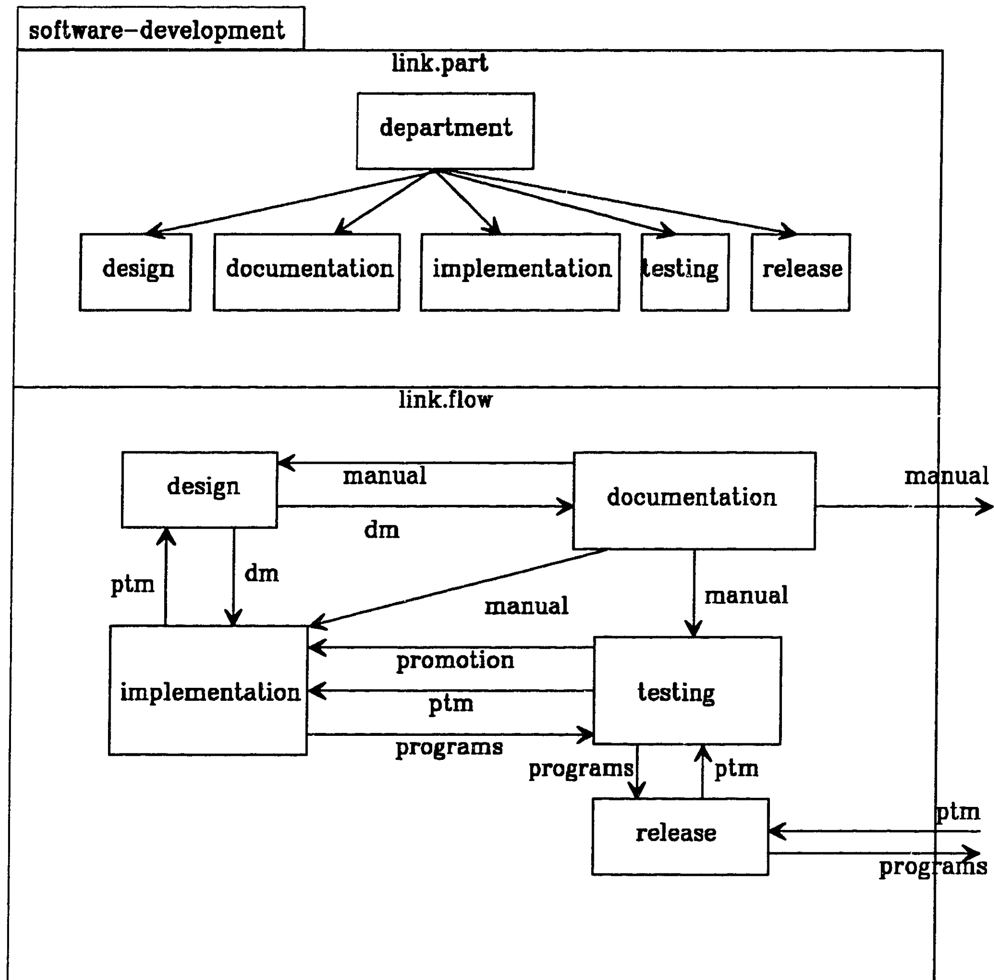


Figure 3.26: Software development organization

Figure 3.27 shows the forms used to coordinate action between all the software development models.

ptm	
program	
description	

promotion	
from	
to	

version–creation	
program	
base–version	
test–model	
new–version	

Figure 3.27: These are the some of the forms which are used to coordinate the action within the software development organization. The PTM (program trouble memo) form is used to report errors. The promotion form is used to promote systems to higher test and release levels. The version-creation form is used to create new versions of programs.

Chapter 4

Organizational Structure

Configurators are bound together structurally with the use of links. Links come in various types, as shown in figure 4.1. A link type represents a structural aspect of an organization. Relationships between configurators which are not explicitly represented by links are implicitly implemented by the message passing actions of a configurator. There are multiple types of links.

A link of type part is a link which connects two configurators, where one configurator is a subpart of another. A chain of part links forms a nested structure of parts. When Ubik reasons over an organization's structure, it can transitively follow part links.

A link of type prototype is a link which connects two configurators, where one configurator is a subclass of another. The subclass configurator uses its prototype's structure as a default when it is created. An organization is continually changing. To reflect this change, all links between configurators can be changed. Prototype links tend to be the most stable type of links, but even they can change. For example, an organization might introduce a new class of employee, such as service employee, and reclassify some of its existing employees to the new class.

A link of type flow is a part link which designates a message which can flow along it. Messages can be sent explicitly by one configurator to another. A flow link represents a more stable relationship between configurators, where all messages specified in the flow link will automatically flow from one configurator to another.

A link of type value is a part link which specifies for a configurator that the attached configurator is to be interpreted as its value. A part link is usually interpreted as an attribute. A value link is then interpreted as the value of the attribute. For example, if an employee has a part link of salary, then the value link attached to salary would be the actual salary. Value links are specified using a dot notation. A salary of 30000 would be written as salary.30000.

A link of type tapeworm specifies for a configurator all the other configurators which are monitoring or censoring it. Tapeworms maintain an

organizational network by monitoring the network's action and constraining its modification.

A link of type sponsor controls the action of the configurators within an organizational network. Each configurator which can take action has a link to a sponsor, which gives it the computational power to carry out the action. The relationship between the sponsors and configurators determines the execution speed of the parallel acting configurators which in turn determines the organization's focus of attention.

A link of type batch specifies for a configurator the configurators which it has received as messages but cannot yet process. When a configurator receives a message, it can execute the message, reject the message, or batch the message.

All links can be labeled. A label specifies the name of a link. Questers which can reason over networks of configurators use the labels to process subsets of links.

4.1 Configurators

A configurator is the basic computational object in Ubik. It takes action by sending and processing messages. It is triggered into action when it receives a message. It is composed of multiple sections, as shown in figure 4.1. The sections are name, input, output, actions, to, control, and link.

The name section identifies the configurator. A configurator only needs to be named if it will be the explicit target of a message. A configurator can be the indirect target of a message with the use of flow links.

The input section specifies the configurators which this configurator is prepared to accept as messages. A configurator will reject messages sent to it which do not appear in the input section. The input section can contain an *and* expression of configurators. The configurators which are received and only partially satisfy the *and* expression are batched. A configurator is fully specified by name, or partially specified with the use of variables.

The output section specifies the messages that the configurator will send or return.

The action section specifies the operations and expressions which will be executed when the configurator is triggered with the receipt of an input message. The most typical action is the relating of the input messages to the output messages.

The `to` section specifies the configurators to which this configurator will be sent when it is triggered into action. Variables can be used to partially specify the destination configurators. The variables are bound when the configurator arrives at its destination. Questers are configurators which use the variables to reason over the organizational structure.

The control section specifies the configurator's execution types and power. The execution type of a configurator are normal, distributed, tapeworm, and

constructor. The power of a configurator is the number of cycles it has for execution.

The link section specifies the links used by this configurator to reference other configurators. There can be a separate link section for each type of link. A section without a name is a link.part section by default.

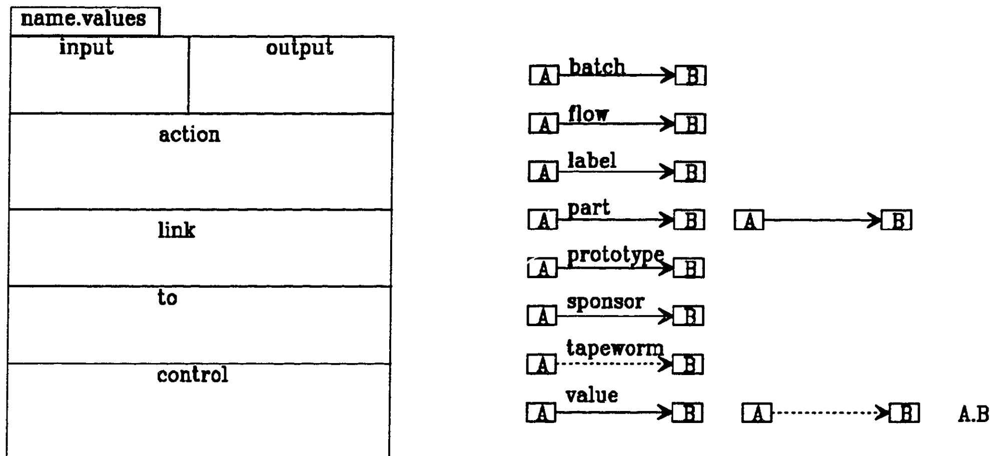


Figure 4.1: Ubik configurator and link types. An organization is a network of linked configurators.

A frame is a structure in a knowledge base system which has a name and slots. The slots have values and facets. The facets are constraints on the values which can be inserted in the slots, and specify attached procedures which are triggered when a value is inserted in the slot or a message is received by the frame. A configurator has some similarities to a frame. The example in figure 4.2 shows an employee frame with two slots: mgr and salary. The facets in this example are restrictions and handler. The restrictions are used to specify bounds on a slot value; the handler is triggered when a message is sent to the slot. An instance of a frame is a copy of the frame with the slot values filled in. In this example, the employee frame has an instance name of jill, the mgr slot has a value of mike, and salary slot has a value of 30000. In Ubik, the employee frame is represented by a collection of linked configurators. This example shows two ways to display a configurator and its links: the Ubik linked representation uses explicit part and prototype links; the Ubik configurator representation uses configurator link sections. In this example there are two link sections: link.part and link.prototype. The Ubik correspondence to frame facets will be discussed later, in section 4.3.

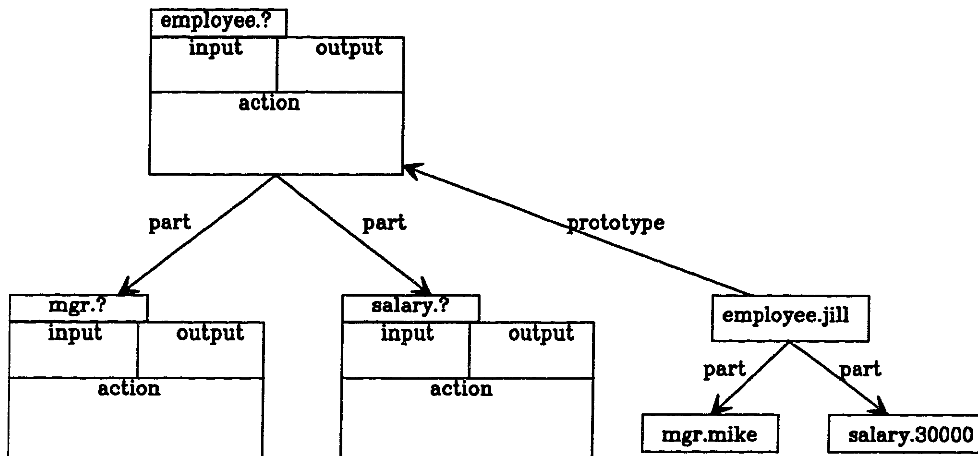
Frame Representation

frame

frame name	*instance name*
employee	?
slots	
name	*facets*
mgr	? restrictions handler
salary	? restrictions handler

instance

frame name	*instance name*
employee	jill
slots	
name	*facets*
mgr	mike
salary	30000



Ubik Configurator Representation

employee.jill
link.part
mgr.mike salary.30000
link.prototype
employee

Figure 4.2: Frame and Ubik representations of an employee concept, with attributes mgr and salary.

4.2 Constructors

A constructor configurator is used to construct an organizational network. A constructor configurator consists of the following sections:

1. To - specifies the configurator to which links are added.
2. Link - specifies the links to add.
3. Control - specifies the operation. The constructor operations are insert, delete, and update.

Figure 4.3 illustrates the use of three constructors.

A	B	C
to employee	to employee.jill	to employee.bill
link.value jill	link.value bill	link.value bill
control insert	control update	control delete

Figure 4.3: Configurators can be used to construct an organizational network. The insert configurator will produce employee.jill. The update configurator will change employee.jill to employee.bill. The delete configurator will change employee.bill to employee.

4.3 Tapeworms

A tapeworm is a configurator which can be attached to another configurator. There are two types of tapeworms: monitors which report on an event, and censors which can prevent an event from occurring. A tapeworm is specified by a configurator with the following sections:

1. Input - specifies the configurator to which the tapeworm is being attached.
2. Action - specifies the action which is to be taken when the tapeworm triggers.
3. Control - specifies that this is a tapeworm command.

Figure 4.4 illustrates a censor tapeworm which will ensure that a paycycle is either weekly or monthly. The tapeworm's input section specifies that it is to be triggered when the paycycle value link for an employee configurator is modified. The tapeworm's control section specifies that it is a tapeworm of type censor, and it censors insert, update, and delete operations. The tapeworm's action section specifies that a boolean expression is evaluated when the tapeworm is triggered. If the boolean expression produces a false result, then the operation is censored.

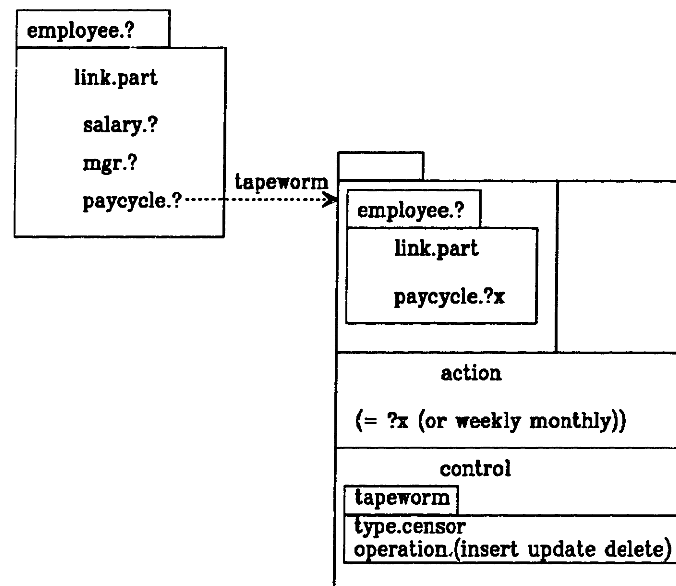


Figure 4.4: The tapeworm paycycle censor ensures that a paycycle has the values weekly or monthly.

Figure 4.5 is a censor which ensures that an employee will not make more than his or her manager. The tapeworm is attached to the salary value link; when triggered, the action section executes as follows. The quester configurator with variable ?e as input sends itself to the employee configurator which caused the tapeworm to trigger. It returns the manager of the employee in variable ?m. The quester configurator with variable ?m as input sends itself to the employee configurator of the manager and returns his salary in variable ?sm. The boolean expression tests whether the manager's salary is greater than the employee's. If it isn't, then the operation is censored.

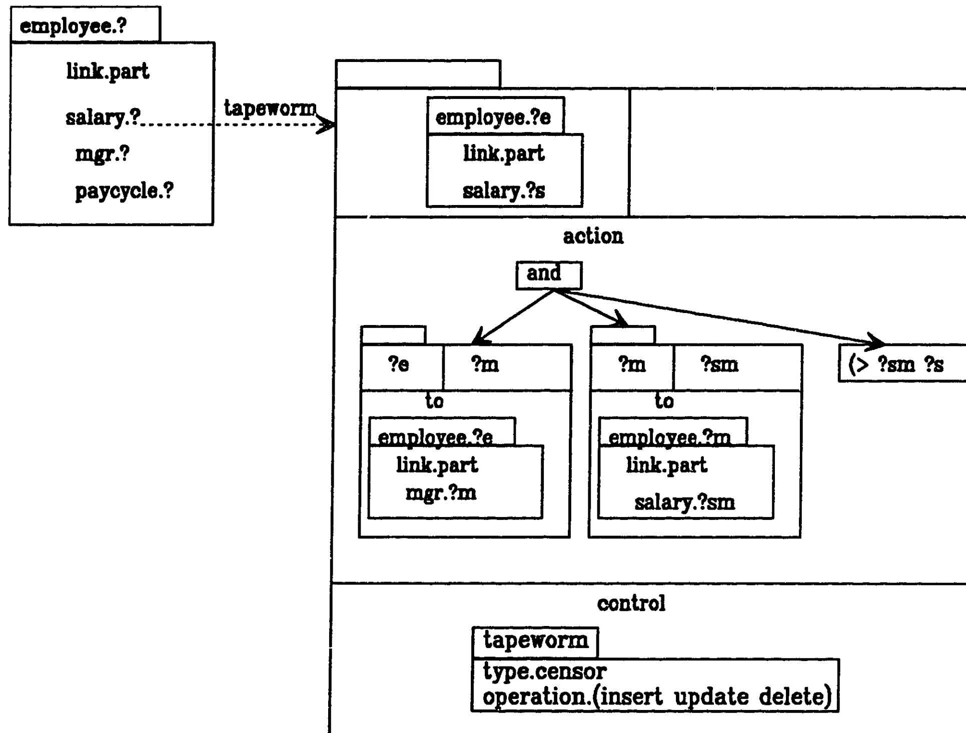


Figure 4.5: The tapeworm salary censor ensures that an employee does not make more than his or her manager.

4.4 Prototypes

A prototype is a concept which *best* represents a collection of concepts. Section 8.5, on prototype development, discusses the construction and recognition of a prototype. A prototype is used within Ubik as follows:

1. To supply default links for the creation of a new configurator. A configurator in Ubik can change its prototype. When this occurs, the new configurator keeps the links it inherited from its prototype at the time of its construction.

2. To supply a class type for the configurators which reference it. When a configurator changes its prototype, it changes its class.

A configurator can have multiple prototypes. Conflicts between links in the multiple prototypes can result in a non-viable configurator.

An example of a prototype is shown in figure 4.6. In this example, the configurator employee is used as a prototype for the manager and non-manager configurators. These configurators inherit the links from the employee configurator. The manager configurator is used in turn by the configurator manager.mike. This configurator inherits all the links from employee and manager. The only links shown are part links, but other types of links, such as tapeworm links, are also inherited.

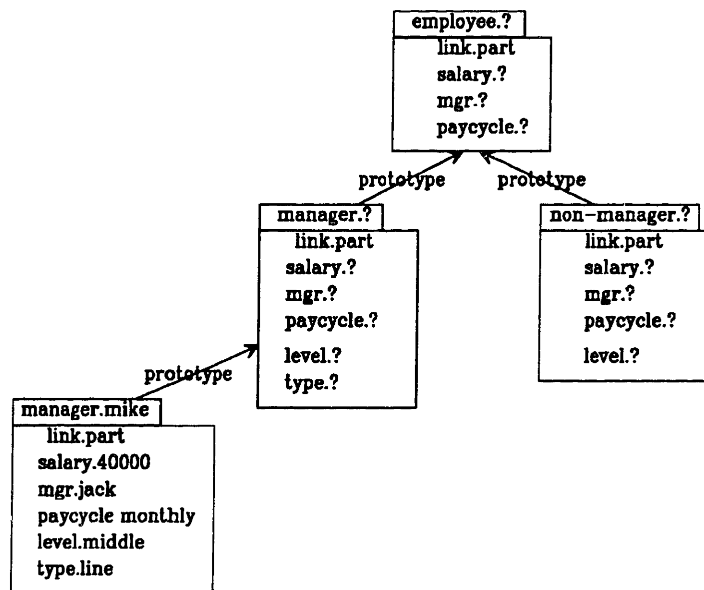


Figure 4.6: The employee prototype hierarchy consists of an employee configurator used as a prototype for the configurator's manager and non-manager, and a manager configurator used as a prototype for the configurator manager.mike.

4.5 Questers

A configurator with a to section is called a quester. The configurator seeks information from the configurators specified in the to section. A quester is used to search the organizational network. An example of a quester has been given in figure 4.4. The action section required two questers, one to find an employee's manager, and the other to find the manager's salary.

Labels can be added to links to increase the amount of information specified by the network. Figure 4.7 shows a network with two sets of labeled part links: label.employee and label.level. The level-hierarchy is constructed using the label.level links. The references to employees from the levels is accomplished by using label.employee links. Figure 4.8 shows a quester using the label links. This quester finds all the employees below top management.

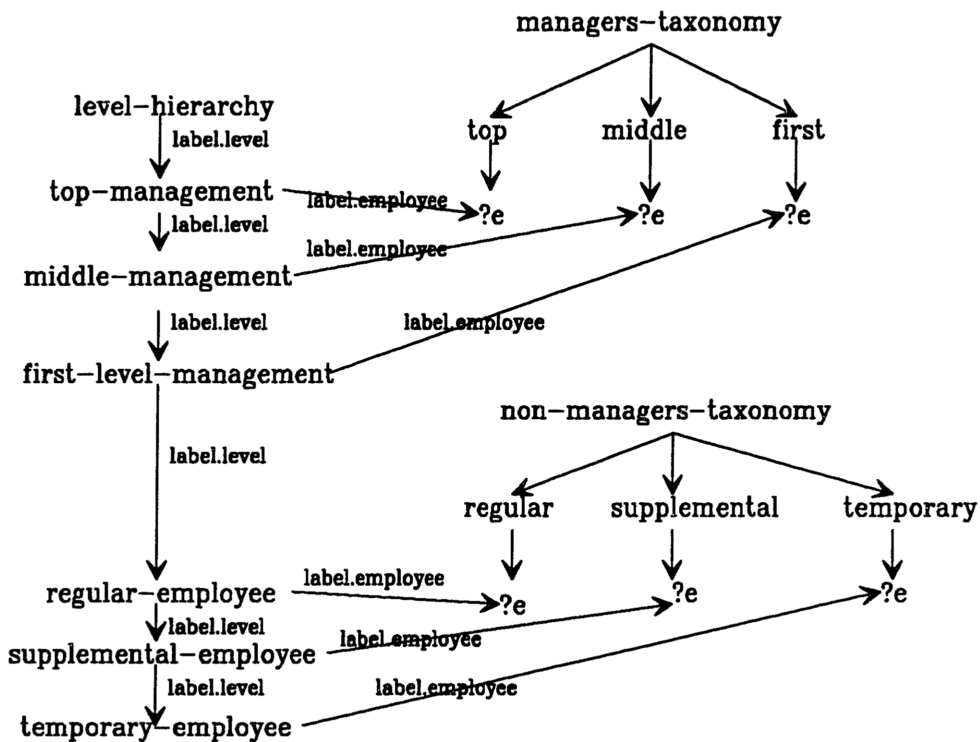


Figure 4.7: Labeled links are used to distinguish between the level hierarchy and the employees who are referenced by the hierarchy.

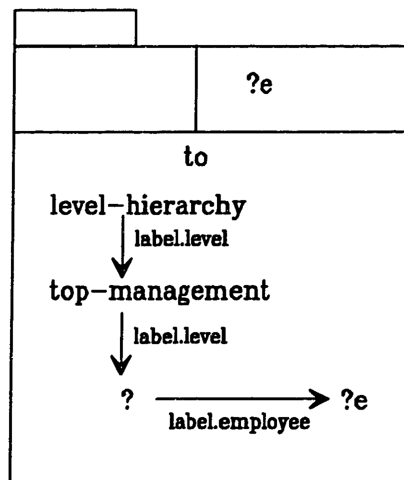


Figure 4.8: The quester examines the structure of the organization to find all the employees below top management.

4.6 Distribution

Configurators can represent the distribution of an organization over multiple locations. All configurators representing an organization have a root link in a distributed configurator. An organization consists of at least one distributed configurator. A distributed configurator is specified by a location command in the control section, as shown in figure 4.9. The control section can also contain a directory command which specifies the other distributed configurators known to this one. Communications between distributed configurators can be by message passing, as shown in figure 4.10, or by link transversal, as shown in figure 4.11.

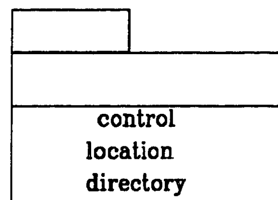


Figure 4.9: A distributed configurator is one with a location command in its control section. The directory command specifies the location of other distributed configurators.

Figure 4.10 shows three distributed configurators: configurator A at location.X, configurator B at location.Y, and configurator C at location.Z. Configurator A sends configurator M1 to B. The location of B is specified in A's directory. It also sends configurator M2 to C. The location of C is not in A's directory, so it is explicitly specified in M2's **to** section.

Figure 4.11 has two distributed configurators, A and B. B is attached to A with a part link. Links can be followed from A to B, or configurators linked to B, without specifying a location. The transversing of the distributed links are automatically handled by Ubik.

All names are local in Ubik. Top-level distributed configurator names are local to their locations. All other names are relative to the configurators to which they are linked. Figure 4.12 shows a distributed configurator which contains multiple occurrences of the same name in different contexts. In the ubik-inc, the name *employee* appears once as the top-level name, and three times in the context of the department configurator. It also appears in the distributed configurators ubik.massachusetts and ubik.california. Each of these appearances is called a context of employee. Each context has a different organizational meaning and appears in different organizational actions. For example, the employees in the sales department have a commission attribute to support the paying of commissions; the employees in the manufacturing department have the shift attribute to support the assigning of personnel to shifts; and the accounting department has the certification

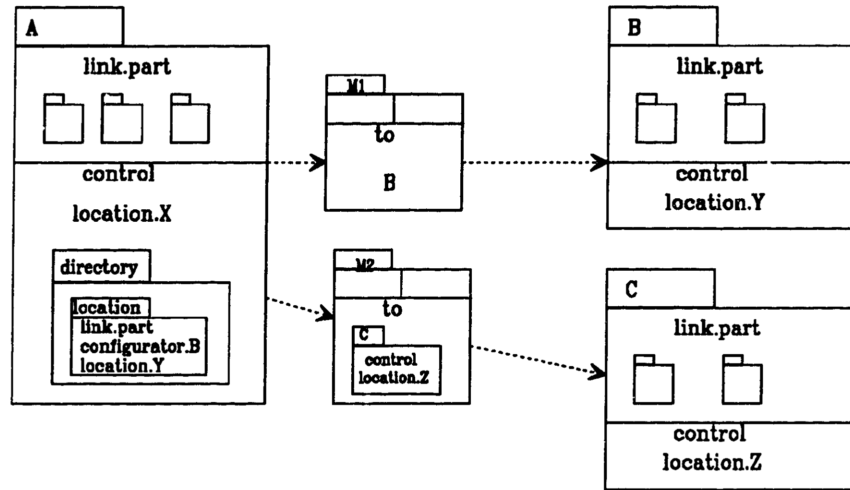


Figure 4.10: Message passing between distributed configurators.

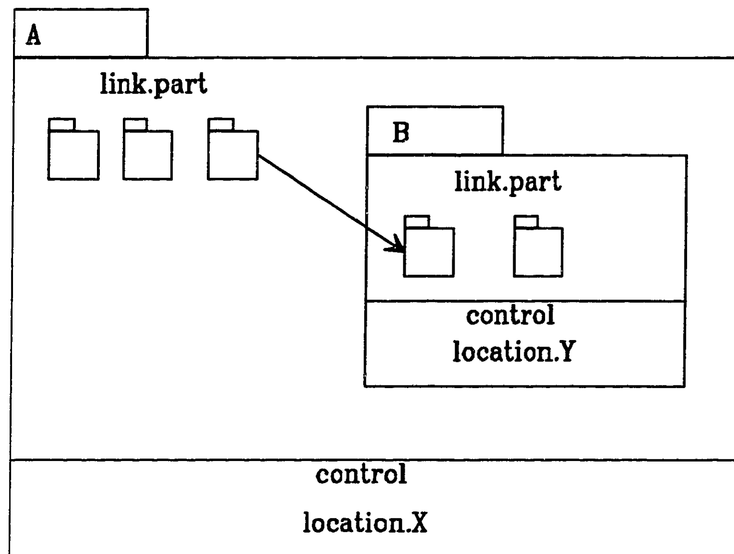


Figure 4.11: Linking between models.

attribute to support federal reporting requirements of that department.

An individual is a concept which has an existence independent of its context. In figure 4.12, the configurators jill, mike, dick, and joe appear in multiple contexts, yet each refers to an individual. In this example, links with the label individual connect all the different contexts of jill.

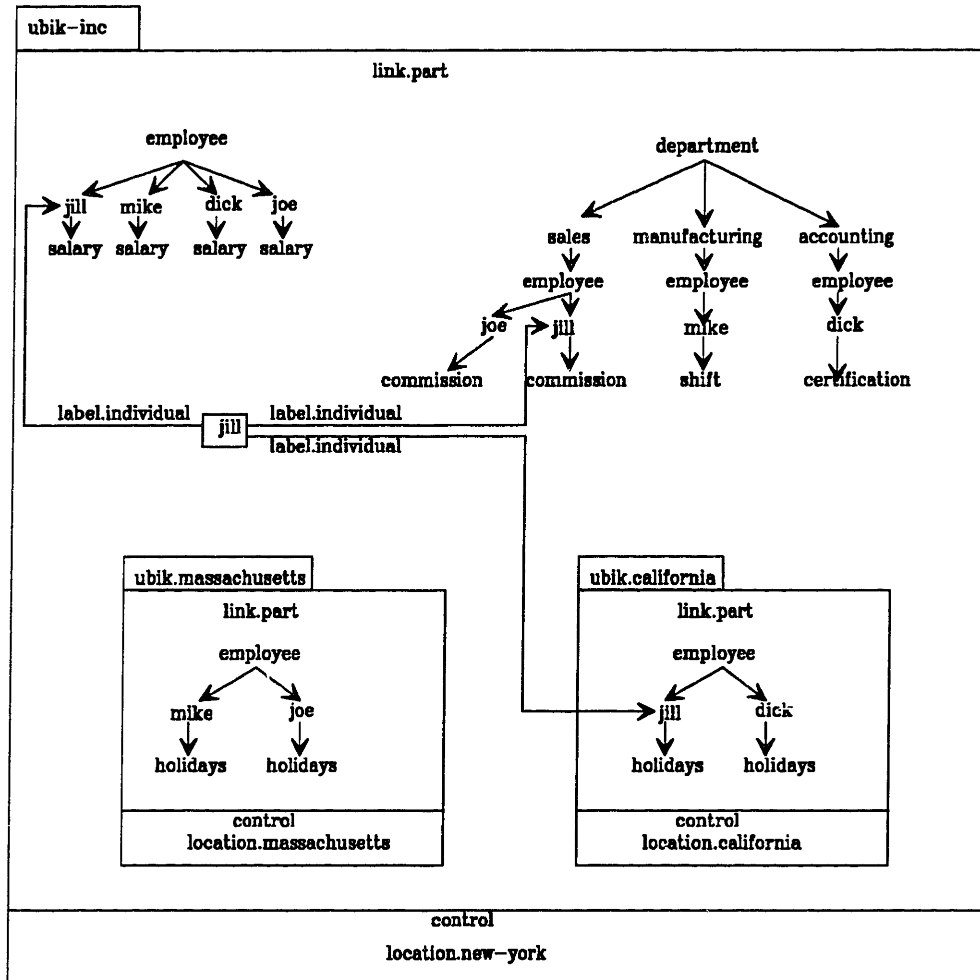


Figure 4.12: Names within Ubik appear within a context. An individual can appear in multiple contexts. Jill is an individual who appears in three contexts. All the contexts are connected by links labeled individual.

4.7 Organizational Concepts

A network of configurators represents a theory of an organization's structure and action. A configurator represents an organizational concept. Ubik's representation has been influenced by the following observations of human concepts:

Concepts are not individual objects; they come in clusters. To Murphy and Medin [54], a cluster of concepts is coherent only if it conforms to a person's naive theory of the world. *Similarity*, according to Murphy and Medin, is insufficient to account for conceptual coherence for the following reasons:

1. It leads naturally to the assumption that categorization is based solely on attribute matching.
2. It ignores the problem of how one decides what is to count as an attribute.
3. It engenders a tendency to view concepts as being little more than the sum of their constituent components.

Some conceptual clusters have no similar features to hold them together. Barsalou [11] shows conceptual clusters which are goal-derived rather than similar. The following objects form a goal-derived conceptual cluster: children, jewelry, portable TVs, paintings, manuscripts, and photograph albums. The goal that defines the cluster is *taking things out of one's home during a fire*.

A theory forms a conceptual cluster according to Carey [18], by constraining induction, explicating causal notions and ontological commitments, analyzing whether terms refer to natural kinds, and specifying aspects of the initial state. A theory is characterized by the phenomena in its domain, its laws and oth... explanatory mechanisms, and the concepts that articulate the laws and the representations of the phenomena. Explanation, to Carey, is at the core of theories. A concept is not a concept in a theory unless it has an explanatory position in the theory. She uses this view to show how children develop concepts between the age of five and ten. For example, the child's concept of eating at the age of five is not a biological concept but a psychological concept. The child associates eating with the ritual of the family dining. At ten, the child associates eating with the biological transformation of food. Eating, although the word is the same, is a different concept at the age of five than at the age of ten, because it has a different explanatory position in a different theory. Similarly, the concept of employee within an organization is a different concept depending on the context in which it appears. The concept of an employee to the personnel department represents a person with a name and address, and a salary. The concept of an employee to the legal department represents a person who falls

under the labor relations law. The concept of an employee to the manufacturing or sales department is a person who participates in the activities of his respective department. Within Ubik, a concept exists within a network of other concepts and actions. The position of a concept within this network determines its meaning to the organization which it is representing.

Organizations support the assembly of resources in order to perform one or more tasks in a distributed and parallel manner. An organization can be thought of as a coherent collection of conceptual clusters. The division of labor within an organization permits experts in different fields of endeavor to work together on common goals. This notion of the coordination of multiple experts is the basis of Minsky's *Society of Mind* [53]. Putnam [58] points out that the idea of division of labor extends to the meaning of concepts in multiple minds within a community. A concept is a sociological idea which requires the cooperation of multiple individuals. *Gold* is an example of a distributed concept. Each member of a community is said to know what gold is if they acquire the stereotype of gold for that community. Putnam calls obtaining this stereotypical concept *linguistic obligation*. The linguistic obligation for gold in our community might only be that it is yellow, valuable, and used in jewelry. The *division of labor* of the concept of gold occurs because there are members of the community who have a greater knowledge of the concept of gold: there are assayers who can determine if something is gold, there are financial experts who can tell the monetary value of gold, and there are jewelers who can fashion gold into jewelry.

Chapter 5

Action

Organizational action is both interactive and cyclic. The interactive actions consist of unscheduled, although not necessarily unplanned, activities. Examples are: order entry, credit checks, and ad-hoc queries. Cyclic actions are the scheduled activities. Examples are: the closing of books and other accounting cycles, invoice billing, payroll, and scheduled reports. Action is initiated by the receipt of a message, and all the actions are executed in parallel with each other. A configurator, on the receipt of a message, performs one of the following:

1. Execute - the message is accepted and processed by the action.
2. Reject - the message fails to match an input pattern associated with the action.
3. Batch - the message cannot be currently processed. It is batched until it can be processed.

Action can be performed explicitly with the receipt of a message by a configurator, or implicitly with the interception of a message by a tapeworm. Messages can be sent explicitly to configurators or can be implicitly sent using flow links.

Ubik action is based on the actor model of computation. Actors are a computational model pioneered by Carl Hewitt [37,40]. The message passing semantics group at MIT, under Hewitt's direction, has developed the theory [9,20,36] and implementation [43,48,49,51] of actor systems. An actor is a computational object with the following properties:

1. Contains references to other actors. It can dynamically obtain additional actor references.
2. Accepts a message which can cause it to change its state. A change of state is accomplished by changing its references.
3. Sends a message asynchronously to an actor it references.

Actors execute in parallel with each other. Synchronous message sending, such as calls to subroutines, requires a collection of actors which communicate using continuations [23,40]. Actors can be serialized or unserialized. A serialized actor locks itself when it receives a message. All subsequent messages are queued until the actor unlocks. A serialized actor is used for actors which change their state, such as bank accounts. Unserialized actors can always accept messages. An unserialized actor is similar to program functions which have no internal state.

5.1 Message Passing

The most basic communication in Ubik is the sending of a message asynchronously, as illustrated in figure 5.1. In this example, configurator A sends configurator M2 as a message to B. A flow link can be used to specify default paths for messages leaving a configurator. The example in figure 5.2 uses a flow link to specify the sending of all M2 messages from A to B.

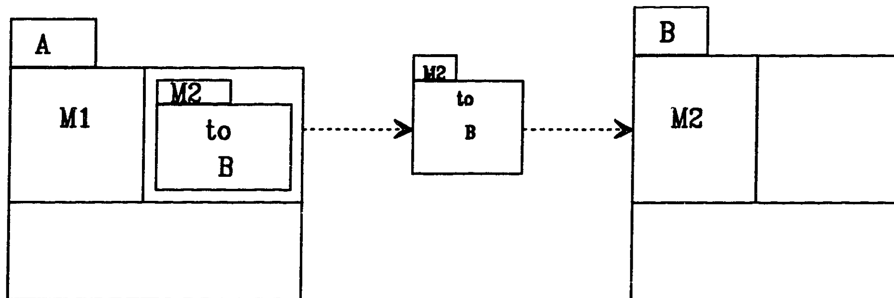


Figure 5.1: Configurator A sends message M2 to configurator B.

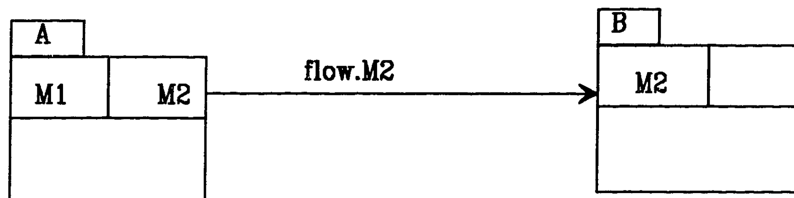


Figure 5.2: A flow link is used to specify that all messages from configurator A are to be sent to configurator B.

Synchronous communication is the sending of a message and waiting for a reply. Replies are handled automatically by the Ubik system if a variable in the output section of a nested configurator is referenced within the

outer configurator. Figure 5.3 illustrates the explicit specification of a reply by an end-user with the use of the reply command in a configurator's control section. In this example, the output operation sends a purchase-requisition to the purchasing department and expects a purchase-order in reply. The purchase-requisition contains a reply command which specifies that a purchase-order is to be returned with the purchase-order value link bound to the variable ?x. The department.purchasing binds the puchase-order.1209 to the output configurator pattern purchase-order.?po. Ubik unifies the reply configurator pattern with the output configurator pattern, resulting in the variable ?x being bound to the value 1209.

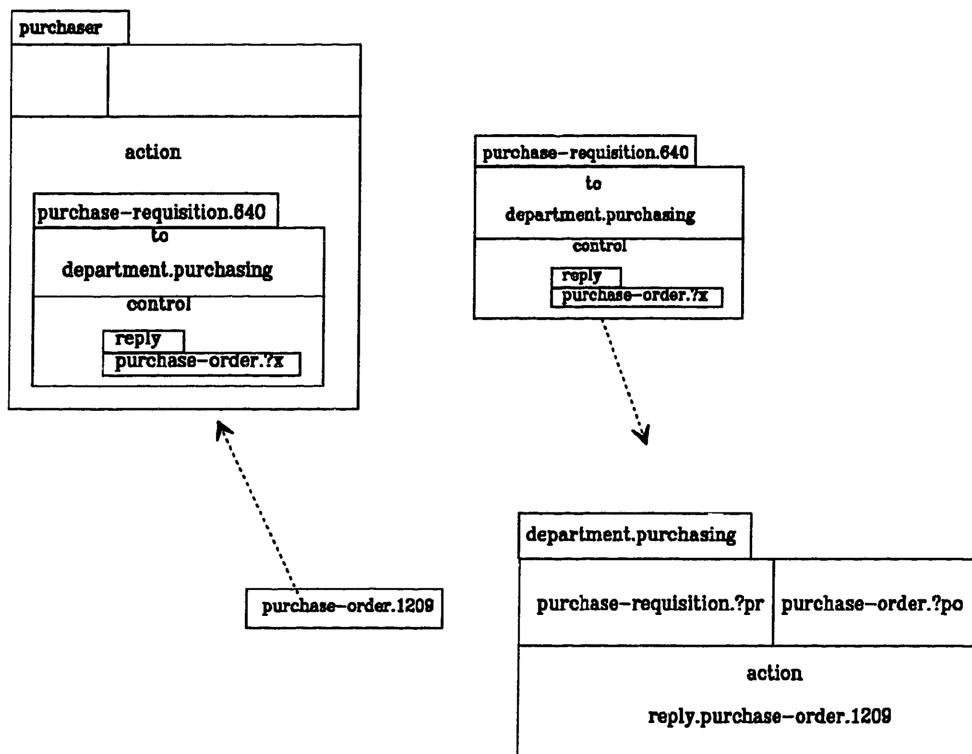


Figure 5.3: The purchaser sends a purchase-requisition message to the purchasing department. The purchasing department replies with a purchase-order.

Communications can be sent to multiple destinations, as illustrated in figure 5.4. In this example, the notification-of-receipt message is sent to the purchasing and accounting departments.

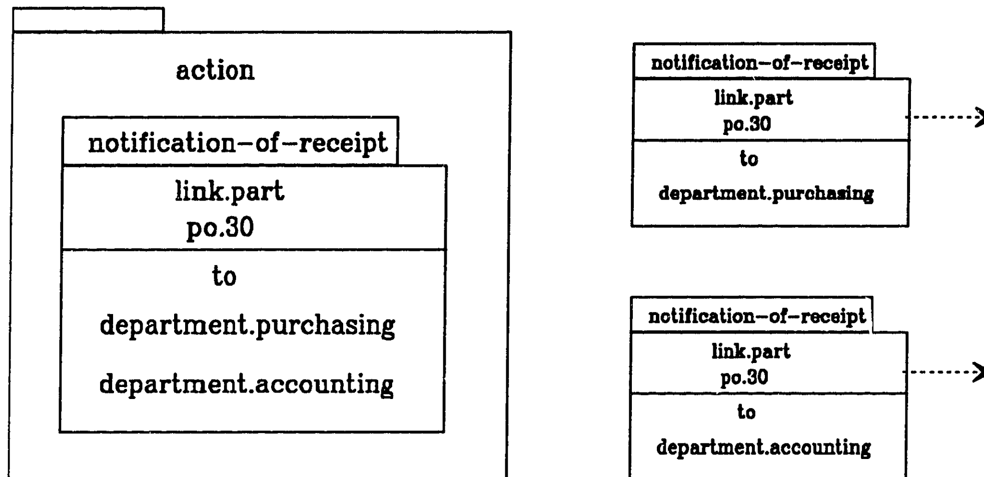


Figure 5.4: The notification-of-receipt message is sent to the multiple destinations of department.purchasing and department.accounting.

5.2 Or Parallelism

Or parallelism is the triggering of more than one configurator by the receipt of a message. An example of *or* parallelism is shown in figure 5.5. In this example, the employee message is received by the assign-employee configurator. The message is sent to the legal and personnel configurators, which are flow linked to assign-employee. Both these configurators trigger when they receive the message.

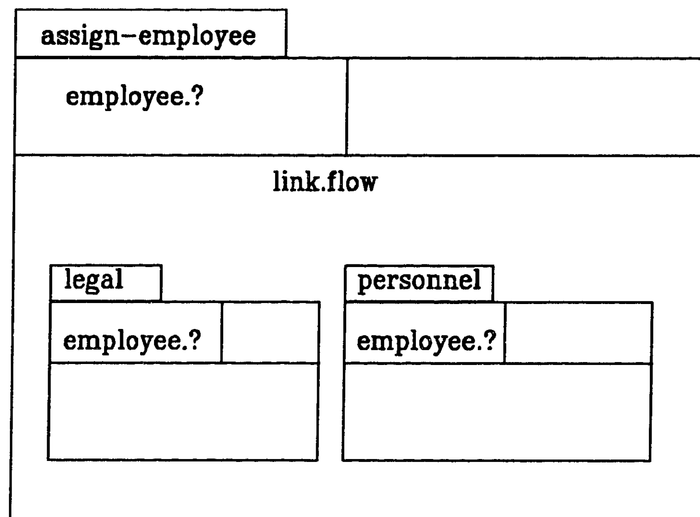


Figure 5.5: Or parallelism is the receiving of the same messages by multiple configurators.

5.3 And Batching

Applications frequently require the receipt of multiple messages before an action can take place. Batching is specified by an *and* expression in a configurator's input section. All the configurators in the *and* expression must be matched before the configurator will trigger. Figure 5.6 illustrates the use of *and batching.* In this example, a receiving-ticket and an item-received with the same purchase-order number trigger the department.shipping configurator. Unmatched input messages are automatically linked to the configurator with a batch link by the Ubik system. In this example, the receiving-tickets with purchase-order numbers 10, 11, and 12 are waiting in the batch for item-received messages with those numbers.

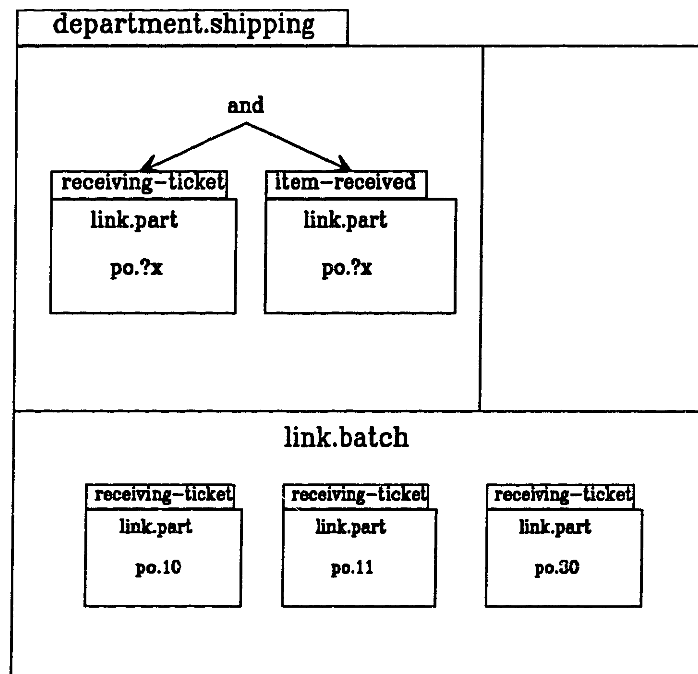


Figure 5.6: And batching is used to delay the sending of the receiving-ticket until a matching item-received form has arrived.

5.4 Bureaucratic Paths

A bureaucracy contains multiple layers of middle management which control the flow of messages and tasks. Bureaucratic paths are flow links attached to configurators. The configurators take bureaucratic action by redirecting the message flow. In figure 5.7, the flow.employee-message link feeds employee-messages into the exceptional-employees configurator. This configurator sends the employee-message to all the employees who make more than their managers. It operates as follows:

1. The quester with the empty input section and the output section with variables (?emp, ?s, and ?m) finds all the employees with their manager and salary.

2. The quester with the input section with variable ?m and the output section ?sm finds the salary for each manager.

3. The boolean tests the employees' salary against the managers'; only the employees whose salary are greater than their managers will remain bound to the variable ?emp.

4. The input message which is in variable ?message is sent to all the employees specified in variable ?emp.

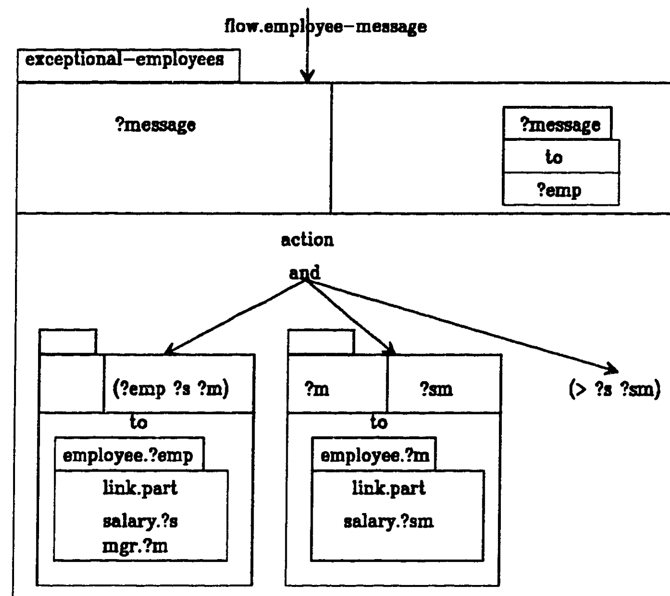


Figure 5.7: A bureaucratic path is a flow link which feeds into a configurator, which redirects the message flow. In this example, the exceptional-employees configurator sends the message to all the employees who make more than their manager.

5.5 Serializers

A serialized configurator locks itself when it receives a message. It is unlocked when it is updated or explicitly unlocked with an unlock operation. A serialized configurator is specified with a serialize command in its control section. Figure 5.8 shows a bank account application. The bank-account configurator contains four configurators flow linked to it: balance, balance-handler, deposit-handler, and withdrawal-handler. The balance configurator is serialized. The bank-account configurator receives either a balance-request, withdrawal, or deposit message. The message is forwarded along the appropriate link to the handler which will accept it. The handlers work as follows:

1. The input section of the bank-account configurator specifies that the configurator will accept the messages balance-request, withdrawal, and deposit. Its link.part section specifies the balance configurator which will contain the balance for the bank account. It is a serialized configurator. The link.flow section contains the configurators which process the input messages. The input section of the configurators in the link.flow section specify the messages each configurator will process. The output section specifies the variable ?t which will contain the transaction slip. The bank-account configurator's output section returns the transaction slip, as specified in variable ?t, to the configurator which sent the bank-account message being processed.

2. The balance-handler configurator is a quester which is sent to the balance configurator when triggered, and returns its value in variable ?b. The output section creates a transaction slip with the balance.

3. The deposit-handler is triggered by the receipt of a deposit message. Its action section contains an update constructor which will update the balance by the amount deposited. The output section creates a transaction-slip which returns the amount of the deposit.

4. The withdrawal-handler is triggered by the receipt of a withdrawal message. Its action section contains a complex constructor which is sent the to the balance configurator. The action section of the constructor is written in Ubik linear syntax. It contains a let statement which binds the variable ?new-balance to the balance, minus the withdrawal amount. Next, a condition statement is executed as follows:

    (a) If ?new-balance is equal to or greater than zero, then the balance is updated to the value bound to the ?new-balance variable using an update operation. After completion, the update operation unlocks the serialized balance configurator. In parallel, a transaction slip is created by the reply operation. The reply operation binds the transaction slip to the output variable ?t.

(b) If ?new-balance is less than zero, then the balance configurator is unlocked with the unlock operation and the reply operation creates a transaction slip.

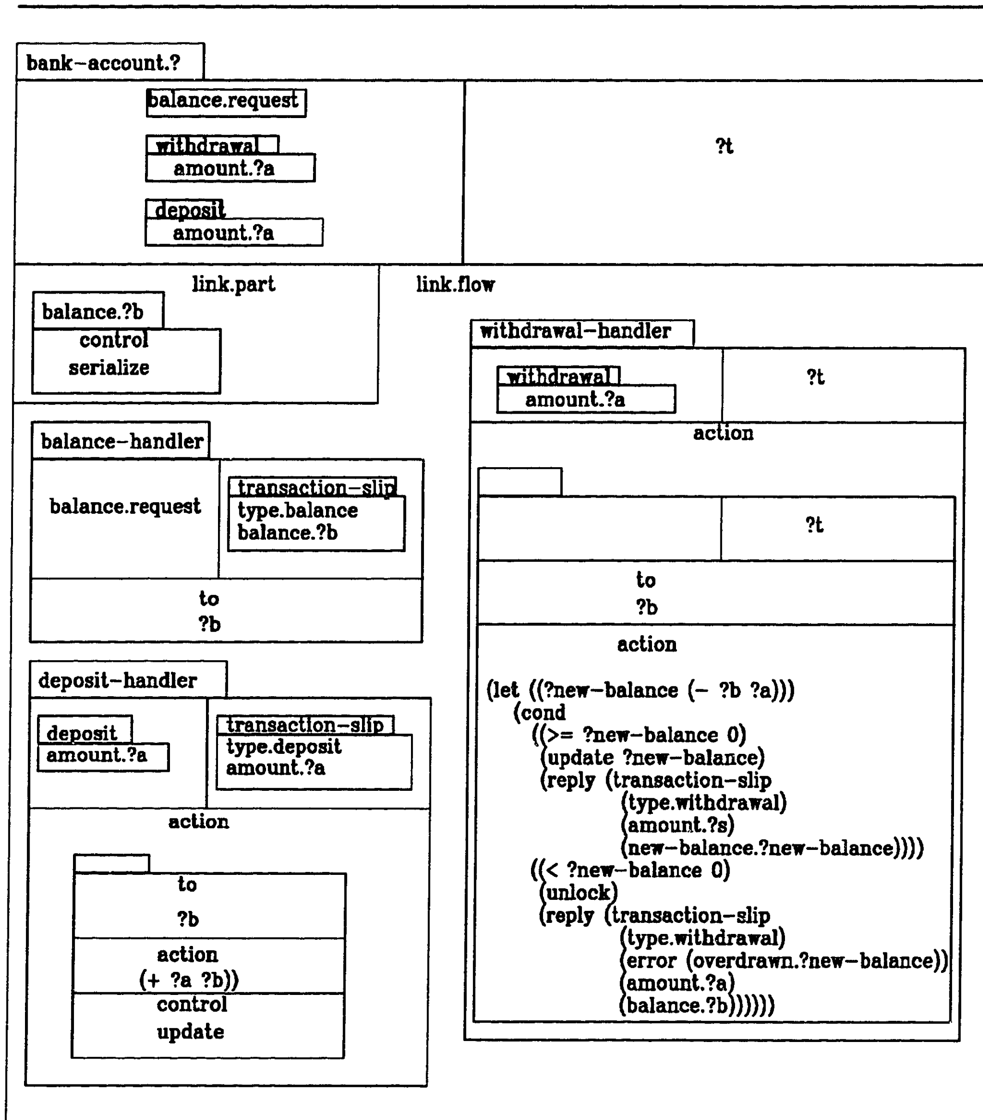


Figure 5.8: The bank account consists of four configurators. The balance configurator is serialized; it will lock when it receives a message.

Chapter 6

Tapeworms

Simon [61] notes the complexity of organizational goals as follows:

> In the decision-making situations of real life, a course of action, to be acceptable, must satisfy a whole set of requirements, or constraints. Sometimes one of these requirements is singled out and referred to as the goal of the action. But the choice of one of the constraints, from many, is to a large extent arbitrary. For many purposes it is more meaningful to refer to the whole set of requirements as the (complex) goal of the action.

Goals in artificial intelligence research usually refer to objects which search for solutions in a problem space. Using this interpretation, the problem space within Ubik is the organizational network. The searching objects are configurators of type constructors, questers, and tapeworms. The search space is constrained by the power conflicts between the search objects as represented by sponsors. The goals of an organization are implicitly represented by the structure of the organization and, given a situation, the organization will tend to react in a way largely predetermined by its structure. Tapeworms add to the organizational structure by providing constraints within and between the configurators which comprise the network.

Tapeworms and questers are used for many purposes within Ubik, some of which follow:

- Monitor the applications.
- Maintain application constraints.
- Censor or replace organizational activity which, if allowed to continue, would cause an application error.
- Seek information or initiate activity throughout the distributed organization.

A tapeworms has been described by John Brunner in "Shockwave Rider" [15] as a computational entity which lays in wait until a condition occurs to trigger it. A tapeworm has a head, which is used to navigate the tapeworm into the part of the organization in which it can take action, and a tail which contains the action. Tapeworms can have both positive and negative effects. To Brunner, a tapeworm is positive in that it provides a way for the hero of his story to blackmail the government into keeping a public interest group on the national computer-net. If the government, in the story, acts against the group, the hero's tapeworm will wipe out important databases accessed though the net. To the government, though, the tapeworm has a negative effect. The communication mechanism used by the tapeworm is the same mechanism used to run nationwide distributed applications. In the biological analogy, a virus (which has a negative effect) uses the same mechanism the body uses to run body-wide applications. Rather than something to be avoided, tapeworms are becoming an important computer processing technique, as organizational applications spread over distributed networks.

A tapeworm is a configurator with a tapeworm command in its control section, as shown in figure 6.1. A tapeworm has the following attributes:

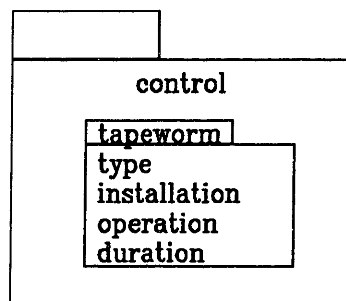


Figure 6.1: Tapeworm

1. **Installation** - how the tapeworm is to be installed. The values are commutative and not-commutative.

2. **Type** - specifies whether the tapeworm is a monitor or censor.

3. **Operation** - the operation which will trigger the tapeworm. The values are insert, delete, update, send, receive, and query.

4. **Duration** - duration of the tapeworm. The values are as follows:

   (a) **Fire** - the number of times this tapeworm will fire.

   (b) **Continuous** - the tapeworm is of continuous duration. This is the default.

   (c) **Time-function** - a time function which specifies the lifetime of the tapeworm.

6.1 Installation

The tapeworm is installed on a configurator within an organizational network. The **to** section specifies to which configurators the tapeworm is to be attached. The input section specifies which configurators are being monitored or censored. In figure 6.2 tape-1's **to** section specifies that the tapeworm is attached to configurator A, and the input section specifies that it is also monitoring any modification to A. Tape-2's **to** section specifies that the tapeworm is to be attached to configurator A, and the input section specifies that it is monitoring configurator A and B. Tape-2 is triggered only when both configurators A and B are modified.

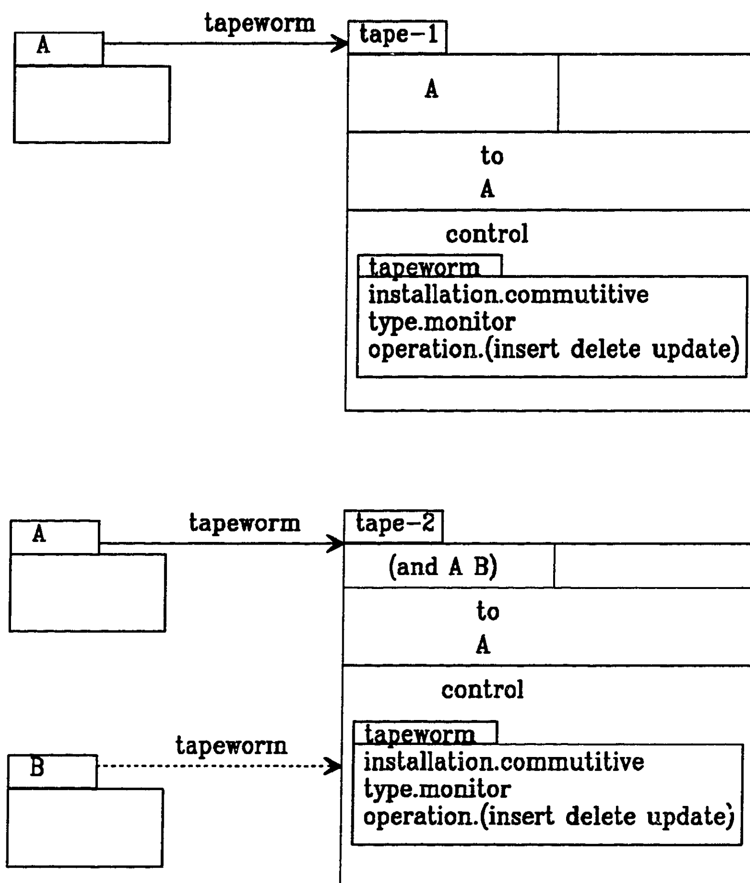


Figure 6.2: Tapeworm tape-1 is attached to and monitoring configurator A. Tapeworm tape-2 is attached to configurator A and is monitoring both configurators A and B.

The concept of commutativity comes from the scientific community metaphor model proposed by Kornfeld and Hewitt [45]. This model describes how communities of distributed scientists communicate. The model is composed of distributed knowledge bases, consisting of predicates and goals, which travel between knowledge bases. Because of the distribution, at any one knowledge base a goal might arrive before a predicate, or a predicate might arrive before a goal. For a non-commutative knowledge base, if the goal arrives before the predicate, it will never be satisfied, even though the knowledge and the goal are in the same knowledge base. In a commutative knowledge base, the goal will be satisfied even if it arrives first. In Ubik, a tapeworm has the installation attribute of either commutative or non-commutative. These are used as follows:

1. **Commutative** - the tapeworm triggers on all the configurators in the taxonomy at the time of its installation.

2. **Non-commutative** - the tapeworm triggers on only the monitored operations which arrive after installation.

Figure 6.3 illustrates the use of this attribute.

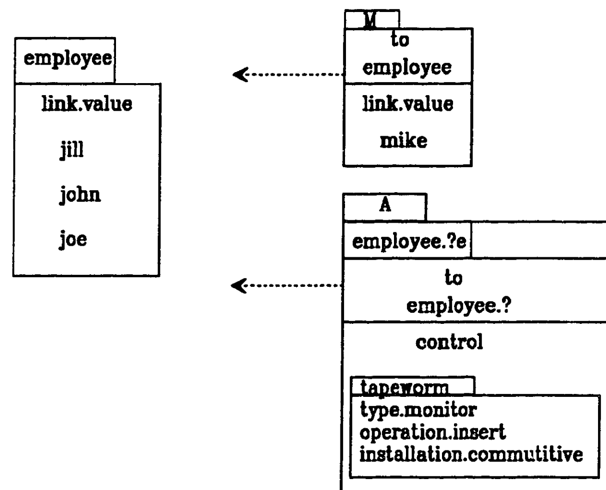


Figure 6.3: The tapeworm is installed with the commutative attribute. If message M arrives before the tapeworm A, or A arrives before M, the results will be the same, that is the tapeworm will be triggered with employees: jill, john, joe, and mike. If the tapeworm has the non-commutative attribute and if M arrives before A, the tapeworm will be triggered with employee mike; if A arrives before M, the tapeworm will not be triggered.

6.2 Types

The tapeworm **type** attribute has the values monitor and censor. A monitor tapeworm is triggered in parallel with the triggering operation. When a censor is triggered, it halts the execution of the triggering operation. The censor's action section is evaluated. The action section contains a boolean expression. If the boolean produces a true value, the triggering operation is allowed to continue; if it produces a false value, the triggering operation is censored.

Censors are used in maintaining organizational constraints and in developing organizational applications. Once an application is operational and being used by multiple other applications, it becomes difficult to restructure the application. An alternative is for the organization to anticipate under what conditions an application is not working, and censor these conditions. Minsky [53] uses this observation to include censors as one of the basic cognitive functions of the mind. As concurrent systems become more complex, and more central to an organization's operations, censors, rather than rewriting, will be used to augment and restrict applications.

6.3 Operations

Operations specify the activity on a configurator which will trigger an attached tapeworm. The operations are as follows:

1. **Insert** - Insert performed by an insert constructor.
2. **Update** - Update performed by a update constructor.
3. **Delete** - Delete performed by a delete constructor.
4. **Query** - Query performed by the quester.
5. **Send** - Sending of a message by a configurator.
6. **Receive** - Receiving of a message by a configurator.

The operation attribute can contain multiple operations. A tapeworm which would trigger on the modification of a configurator would specify the operations insert, update, and delete.

Modern knowledge base systems such as Kee [3] and GoldWorks [1] use multiple types of high-level objects to perform only some of the functions which Ubik performs. By taking a unified approach in supporting the mechanisms of application development, Ubik simplifies the use of the mechanisms, and supports the use of combinations of the mechanisms. The following is a list of knowledge base system concepts, followed by the corresponding Ubik concept:

- forward rule - commutative tapeworm with operations insert, delete, and update.
- backward rule - commutative tapeworm with operation query.
- when-modified daemon - non-commutative tapeworm with operations insert, delete, and update.
- when-referenced daemon - non-commutative tapeworm with operation query.

Ubik supports the following variations of the above concepts not found in either Kee or Goldworks. Some of the variations are as follows:

- rule-insert - commutative tapeworm with operation insert.
- when-receive - non-commutative tapeworm with operation receive.
- when-send - non-commutative tapeworm with operation send.

6.4 Tapeworm and Quester Examples

Some common uses of tapeworms and questers are illustrated in this section. They are freedom of action tapeworms, parasitic tapeworms, and self-propagating questers.

6.4.1 Freedom of Action Tapeworms

A freedom of action tapeworm is the use of a tapeworm to monitor boundary conditions on the operation of configurators. An example is shown in figure 6.4, where the manager of the purchasing department monitors the price the buyers are paying for purchased items. When a buyer purchases an item for over a predetermined maximum price, a message is sent to the manager. The purchasing department is represented by a configurator with the following sections:

1. Input - purchase-requisition which is received by the purchasing department.
2. Output - purchase-order which is produced by a buyer in the department.
3. Link.part - the manager configurator and files configurator are part linked to the department.purchasing. The files configurator has the maximum-item-cost part linked to it.
4. Link.flow - The sections configurator is flow linked to the department.purchasing. The sections configurator has the following sections:

(a) Input - purchase-requisition which is received from the department.purchasing. The purchase-requisition contains a category field. The `to` section of the purchase-requisition form sends the form to the configurator which has the same name as the category on the form.

(b) Output - purchase-orders are sent from the sections configurator.

(c) link.part - the following configurators are part linked to the sections configurator: computers, chemicals, office supplies, and general purchases. These configurators accept purchase-requisitions as input and produce purchase-orders. Each of these configurators has a buyer configurator flow linked to it. The buyer accepts purchase-requisitions and produces purchase-orders.

The freedom of action tapeworm is attached to each buyer in the purchasing department, as shown in figure 6.4. It will trigger when a purchase-order is sent by the buyer.

The freedom of action tapeworm is shown in figure 6.5. It has the following sections:

1. The `to` section specifies that the tapeworm will be attached to all buyers in the department.purchasing. The configurator in the link.part section of the department.purchasing has a name of *?. This is a wildcard which specifies that all the part links emanating from the department will be traced until a buyer is found. In this example the buyers will be found after tracing through the sections configurator and all configurators link.flow connected to it. Since the tapeworm operation is send, the tapeworm will be attached to the output section of the configurator. Any message sent will be bound to the variable ?po.

2. The action section contains an *and* expression with three conjuncts. The configurator with input variable ?po will find each part number and its corresponding unit price. It will return the part number in variable ?pn and the unit price in variable ?up. The configurator with input variable ?pn will look up the maximum cost of that item and return it in variable ?mc. The boolean expression will check if the unit price is greater than the maximum cost. If it is, a copy of the purchase order is sent to the department.purchasing's manager.

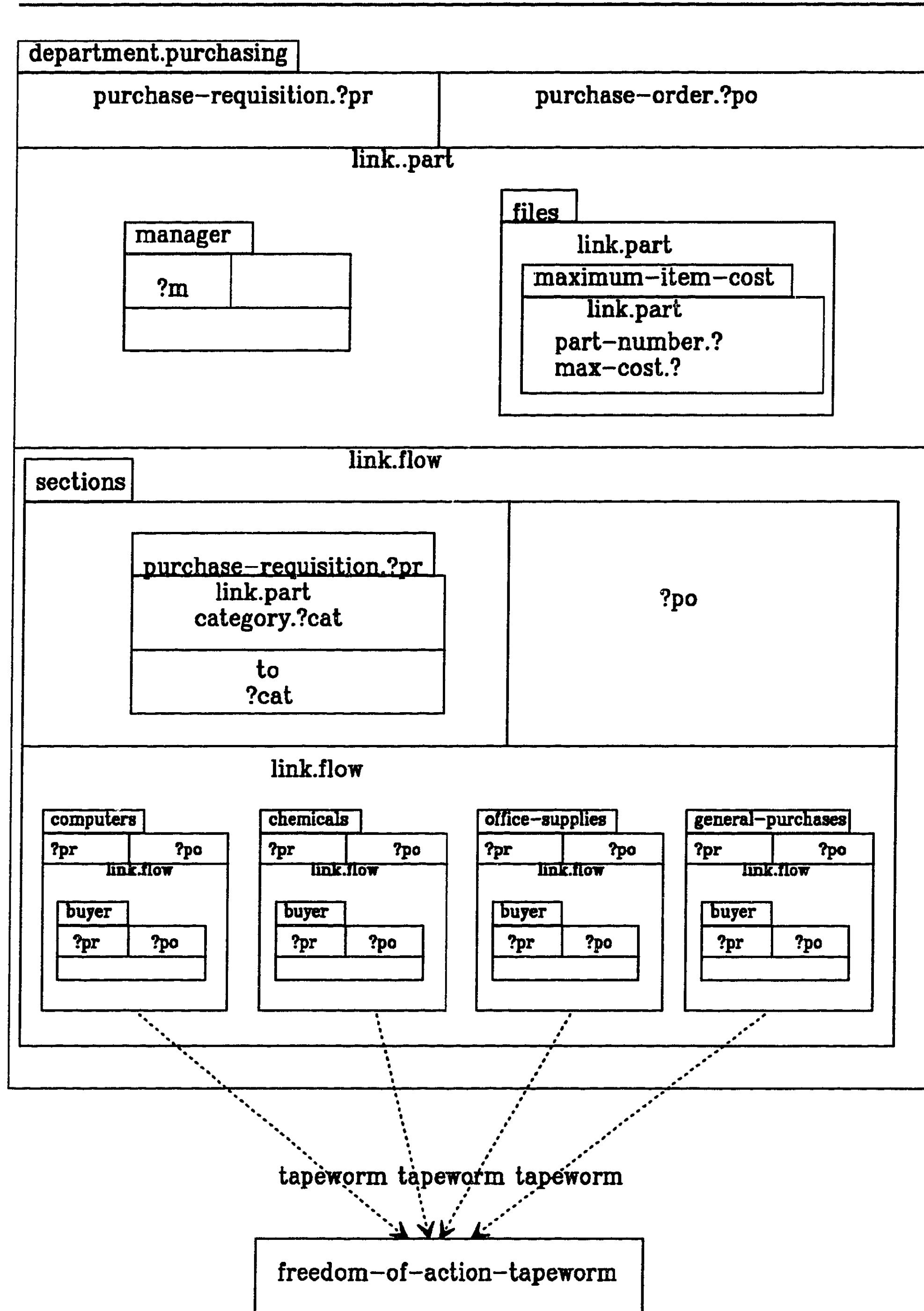


Figure 6.4: Freedom of action tapeworm which will trigger when a buyer in the purchasing department sends a purchase order.

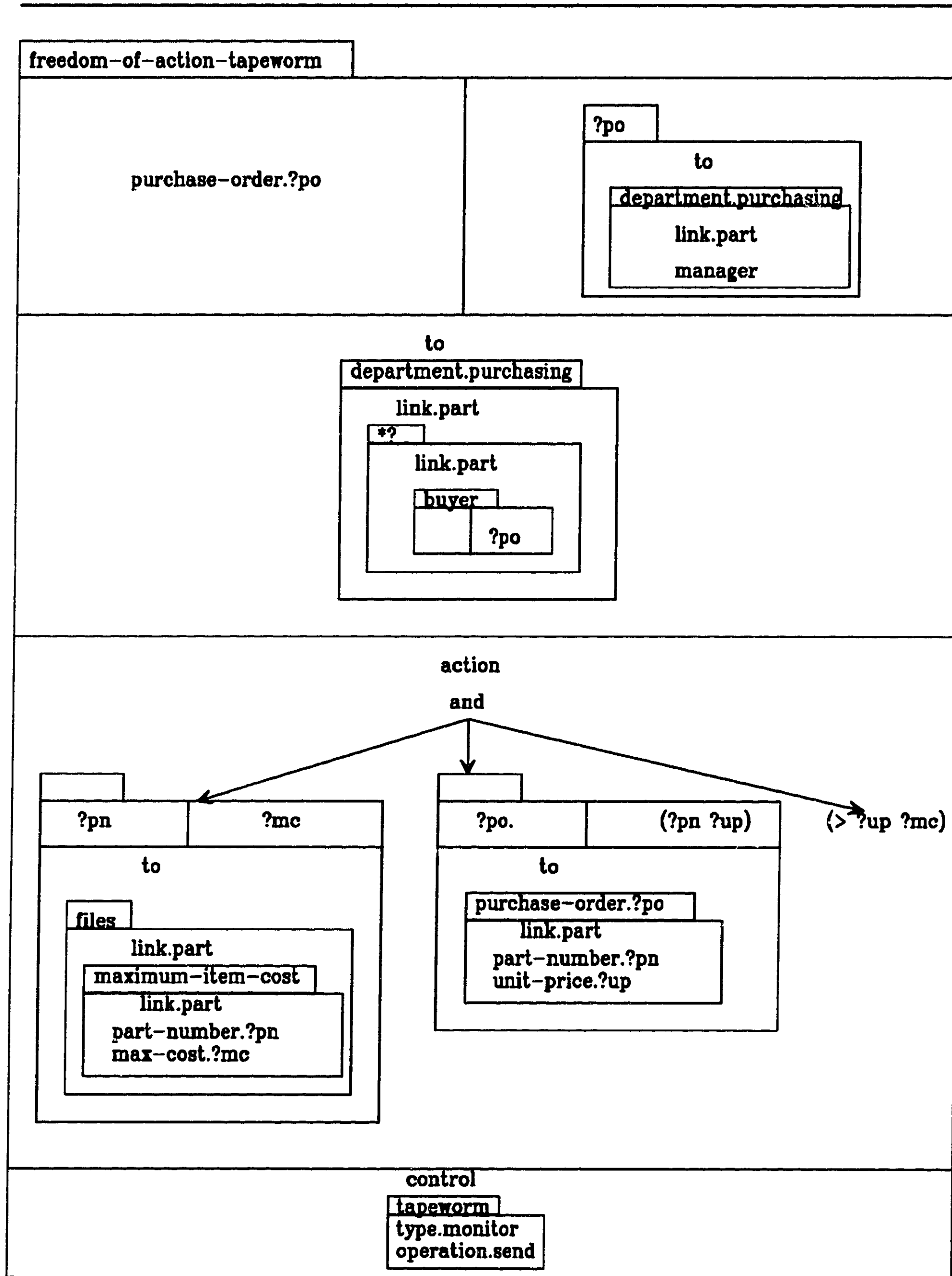


Figure 6.5: This freedom of action tapeworm is attached to the buyers in the purchasing department. It monitors the price of the items the buyer orders.

6.4.2 Parasitic Tapeworm

A parasitic tapeworm is a tapeworm which attaches itself to a configurator which is sent as a message. Figure 6.6 shows a parasitic tapeworm which travels with the purchase-order, triggering when the purchase-order's request date has arrived. The tapeworm has the following sections:

1. The `to` section attaches the tapeworm to all the buyers in the purchasing department.

2. The action section executes when the buyer sends a purchase-order. It contains the overdue purchase order tapeworm. This tapeworm attaches itself to the purchase order being sent. It has the following sections:

   (a) The `to` section attaches the tapeworm to the purchase-order.

   (b) The input section finds the date-requested and buyer fields from the purchase-order.

   (c) The control section contains a timer command. The timer command triggers the tapeworm at the purchase-order date-requested, as specified in the variable ?due.

   (d) The output section sends a copy of the purchase order to the buyer who created it.

3. The control section specifies that the parasitic-tapeworm is of type censor. This tapeworm stops the purchase-order before it is sent, attaches the overdue-purchase-order-tapeworm to the purchase-order, and then allows the sending of the purchase-order to continue.

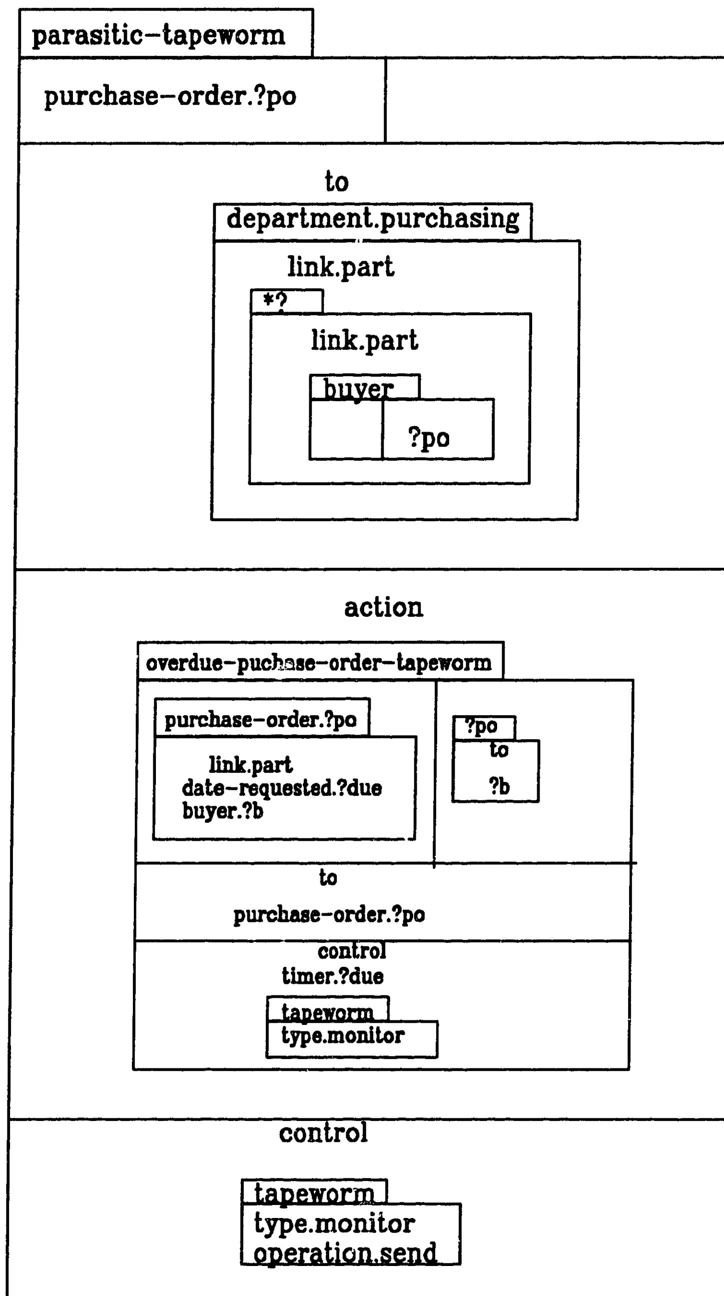


Figure 6.6: The configurator named parasitic-tapeworm attaches itself to the buyers in the purchasing department. When a buyer sends a purchase-order, the action section of the tapeworm contains another tapeworm which installs itself on the purchase-order. It triggers when the purchase-order's due date arrives.

6.4.3 Self-propagating Quester

A self-propagating quester is one which reissues itself when it arrives at its destination. Figure 6.7 shows a self-propagating quester which sends itself to all the distributed configurators contained in the local distributed configurator's directory. When it arrives at its destination, it reissues itself. The action section of the quester operates as follows:

1. The configurator named quester searches through all the configurators in the current location, looking for purchase-orders from the Holland Shade Company.

2. The configurator with output variable ?other-location is a top level distributed configurator, as specified in its control section. The variable ?other-locations is bound to all the locations which appear in the directory.

3. The configurator named find-locations finds the locations not yet visited. It has as input the variables ?visited and ?other-locations. ?visited contains all the locations the purchase-order-quester has already visited. ?other-locations contains all the locations known to the top-level distributed configurator. The configurator produces an output variable ?not-yet-visited which contains the set-difference between the ?other-locations and the ?visited-locations. The configurator also updates the visited configurator with the union of the ?visited and ?other-locations.

4. The configurator named ?poq is bound to the purchase-order-quester configurator. This results in the purchase-order-quester sending itself to all the locations specified in the ?not-yet-visited variable.

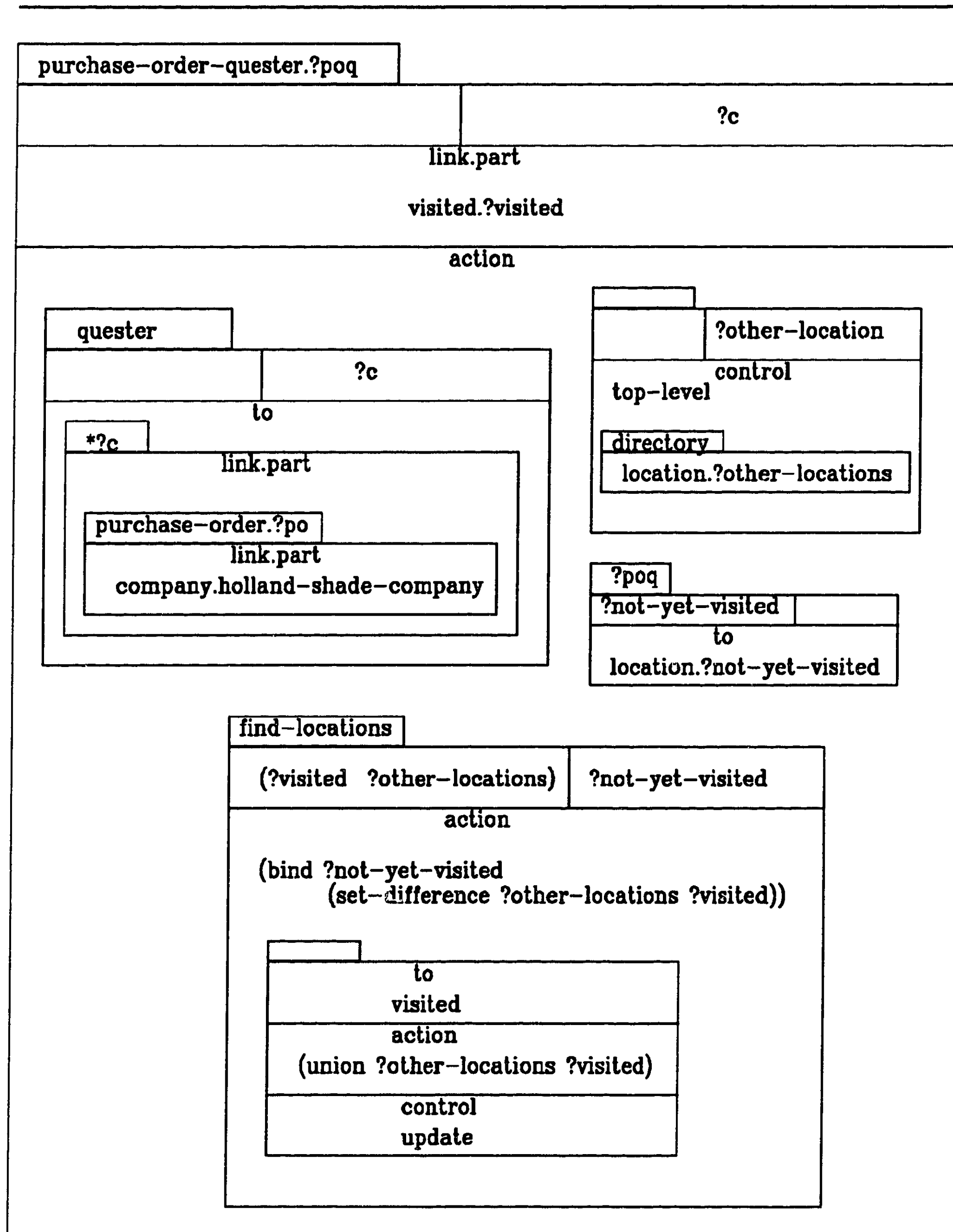


Figure 6.7: The purchase-order-quester locates all the purchase-orders for company.holland-shade-company, and also sends itself to all the distributed configurators it has not yet visited.

Chapter 7

Power

An organizational network can consist of multiple configurators, each capable of executing in parallel. A distributed configurator is one which has a computer network location. Distributed configurators, which reside on different computational devices, can execute in parallel with each other. Parallel execution is also possible within a computer network location, if that location supports computation over multiple processors. Organizations are structured so that applications can execute in parallel. Even thought there is a potential for massive parallel execution within an organization, the parallelism must be controlled for the following reasons:

1. The organization can generate a greater amount of parallel action than there are processors. The parallel action must be mapped to the processors, so that the application's time constraints are met.

2. An organization needs a focus of attention. Even if there is less action available for execution than processors, it might not be appropriate to execute all available action.

Power, according to Scott [59], is the way organizations focus their attention. Scott apportions power among organizational subunits as follows:

1. Subunits that cope more effectively with environmental uncertainty are more likely to acquire power.

2. The lower substitutability of a subunit, the greater its power.

3. The more central and pervasive a subunit, the greater its power.

A Ubik configurator has a processing power given to it by a sponsor configurator. Resource control within Ubik is determined by the relationship of the configurators to their sponsors. The following is some of the information which must be considered when adjusting power within an organization:

1. Pervasiveness of a goal throughout the organization. A goal in Ubik is represented by questers and tapeworms. A goal is pervasive if a quester references many configurators, or a tapeworm is attached to many configurators. A pervasive goal gets more power.

2. Competing active configurators using the same sponsor. The system's power gets divided among the configurators, reducing the power for each one.

3. A configurator which interfaces with the environment can control the message input rate to the organization. If the configurator operates interactively, then it must have enough resources to support real-time message interaction.

Formal power is largely determined by how the organization is separated into subsystems, and how the subsystems are coordinated. Each subsystem has a relatively stable interface which determines what part of the organization and environment affects it, and also what parts it can ignore. March and Simon [52] use the term *bounded rationality* to describe this interface. These subsystems are also said to *satisfice*, in that they are not searching for optimum solutions to their goals, but just acceptable solutions. Once these interfaces are established, the organization has to allocate some of its resources to maintaining them. The competition for power described above is one of the ways in which the organization maintains bounded rationality.

The competition for power also leads to organizational behavior which is not preplanned, but is an emergent behavior of the system from the action of its parts. Adam Smith calls this emergent behavior *the invisible hand*.

A sponsor with linked configurators is shown in figure 7.1. A sponsor is a configurator with a sponsor command in its control section. A sponsor has the following attributes.

1. Regeneration-cycles - the amount of cycles generated in each sponsor regeneration. The cycles are used to determine the processing time of a configurator. The cycles are evenly distributed to all the configurators attached to the sponsor. The cycles are given when the configurator requests its cycles, so that a sponsor does not need to keep track of the location of the configurators linked to it.

2. Regeneration-time - the time between sponsor regeneration. When a sponsor regenerates itself, a new collection of cycles is available for distribution.

3. State - the state of the sponsor. The states are as follows:

   (a) Inhibit - the sponsor will not distribute cycles. Cancellation of configurator action is accomplished by inhibiting the sponsor. The attached configurators will continue operating until they run

out of cycles. When they ask the sponsor for more cycles, they will be inhibited from further action.

(b) Activate - the sponsor distributes cycles.

Each configurator can have a power command in its control section. The power command has the following attributes:

1. Cycles - the amount of cycles the configurator currently possesses.
2. Saturation-level - the maximum amount of cycles that the configurator can possess.

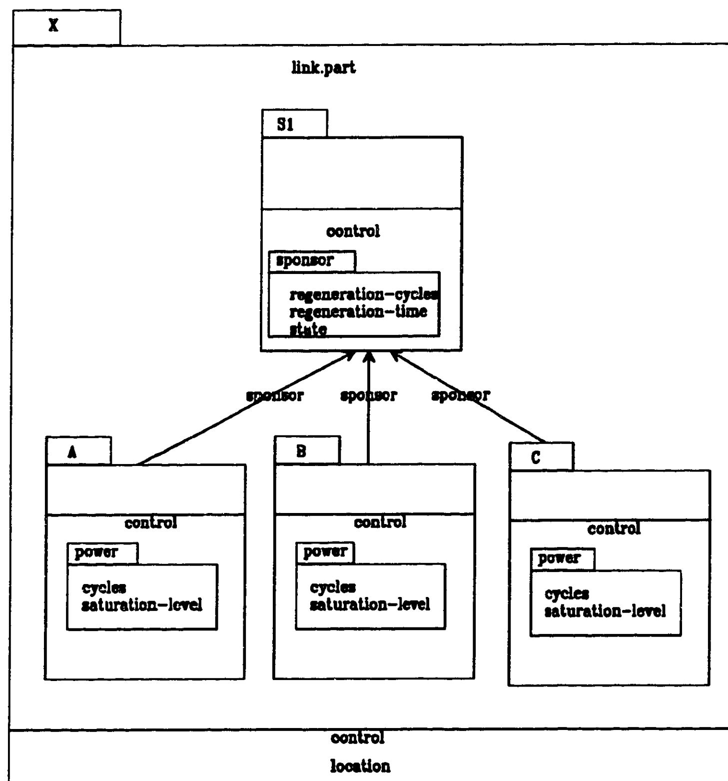


Figure 7.1: A sponsor with linked configurators.

The sponsor model of control was first used by Kornfeld and Hewitt in the scientific community metaphor [45] to simulate the control a sponsoring agency, such as the National Science Foundation, has on directing scientific research. Kornfeld [44] uses sponsors to control parallel reasoning. GoldWorks [1] uses sponsors to control reasoning in a non-parallel expert system. Acore [51] uses sponsors to control an actor based system of parallel executing message passing objects. Ubik uses a revised sponsor model to control organizational power.

The Ubik sponsor cycle works as follows:

1. **Regeneration**: A sponsor gives cycles to requesting configurators. When a sponsor gives up all its cycles, all further requests are batched until regeneration time. At regeneration time, the sponsor replenishes its supply of cycles. Both the supply of cycles for a sponsor and the regeneration time are adjustable. Also, which sponsors can regenerate cycles is adjustable.

2. **Fair Distribution**: During a regeneration cycle, a sponsor will only honor one request from a configurator to process a message. All subsequent requests will be batched until the next regeneration cycle.

3. **Delegation**: Some sponsors cannot regenerate cycles; these sponsors will delegate the configurator's cycle request to another sponsor. Configurators attached to delegation sponsors will probably have less power than configurators attached to regeneration sponsors, because they will have more configurators to compete with for cycles.

4. **Cancellation**: A sponsor can take a request to cancel a configurator. This cancellation request will result in the sponsor sending a cancel message to the canceled configurator when the configurator requests cycles for processing a message. The cancellation request will only be in effect for a specified period of time.

7.1 Sponsor Structure

A sponsor structure consists of a hierarchical collection of sponsors and configurators, where the nodes are sponsors and the leaves are sponsors and configurators. Sponsors in a sponsor structure both regenerate cycles and distribute cycles. Not all the sponsors are regenerating sponsors, but all the sponsors distribute cycles. The distribution of regenerating sponsors in a structure determine how the sponsor structure allocates power.

Centralized sponsor control, as shown in figure 7.2, consists of the sponsor at the head of the structure regenerating cycles, and the rest of the sponsors distributing the cycles. Power in this type of structure is authoritarian, where the all the action is subservient to the head sponsor. In this example, sponsor S1 generates 20 cycles. 6 cycles are distributed to sponsors S2, S3, and D. 6 cycles are distributed to A. 3 cycles are distributed to B and C.

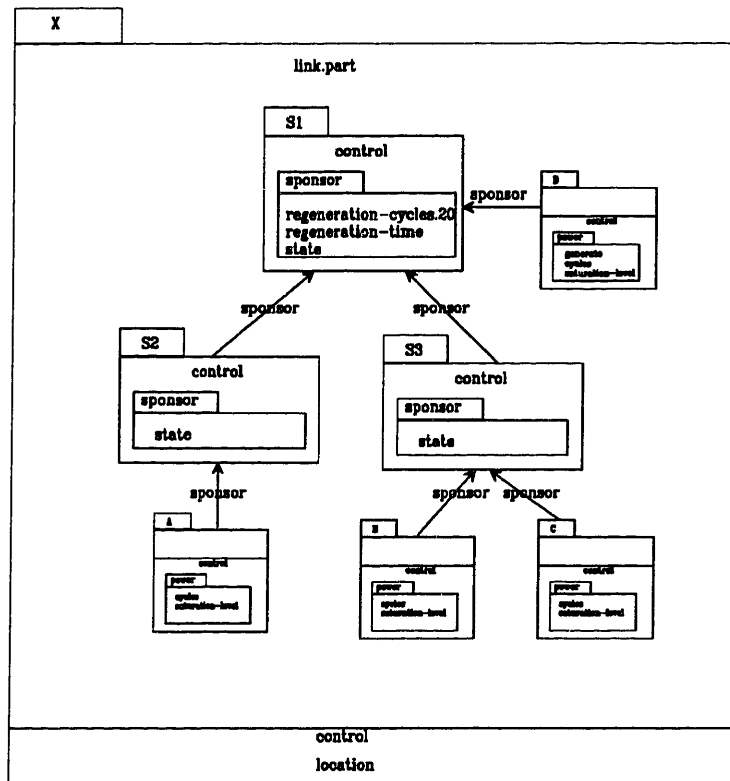


Figure 7.2: Centralized sponsor control consists of one central sponsor giving out cycles to all the actions and configurators in its structure.

Decentralized control, as shown in figure 7.3, consists of each action having its own sponsor. Formal power in this type of control is evenly distributed. In this example, configurators A, B, and C each receive 10 cycles.

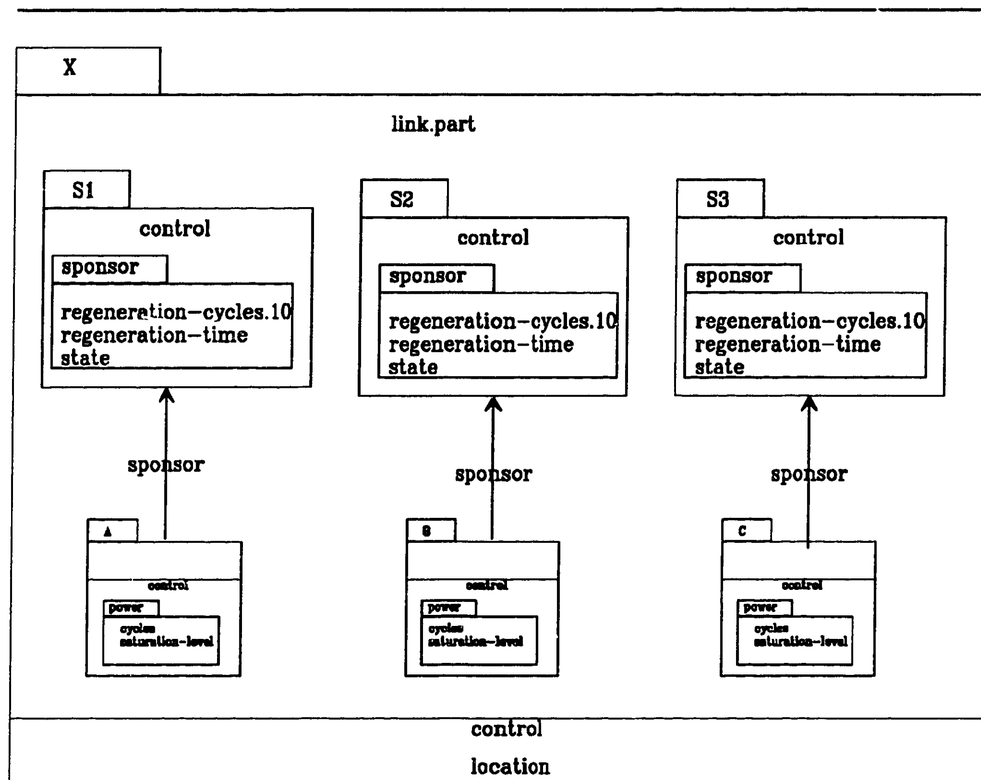


Figure 7.3: Decentralized control with the use of multiple sponsors. Each of these actions gets the full cycles of its attached regenerating sponsor.

Coordinated sponsor control, as shown in figure 7.4, is achieved by having multiple regeneration sponsors in the sponsor structure. This is the most flexible type of control, in that any sponsor in the structure can be dynamically made into a regeneration or non-regeneration sponsor, in order to adjust the organization's actions to its needs. In this example, 6 cycles are given to S2, S3, and D. 6 cycles are given to A. S3 generates 10 cycles in addition to the 6 given to it from S1; therefore, B and C receive 8 cycles each.

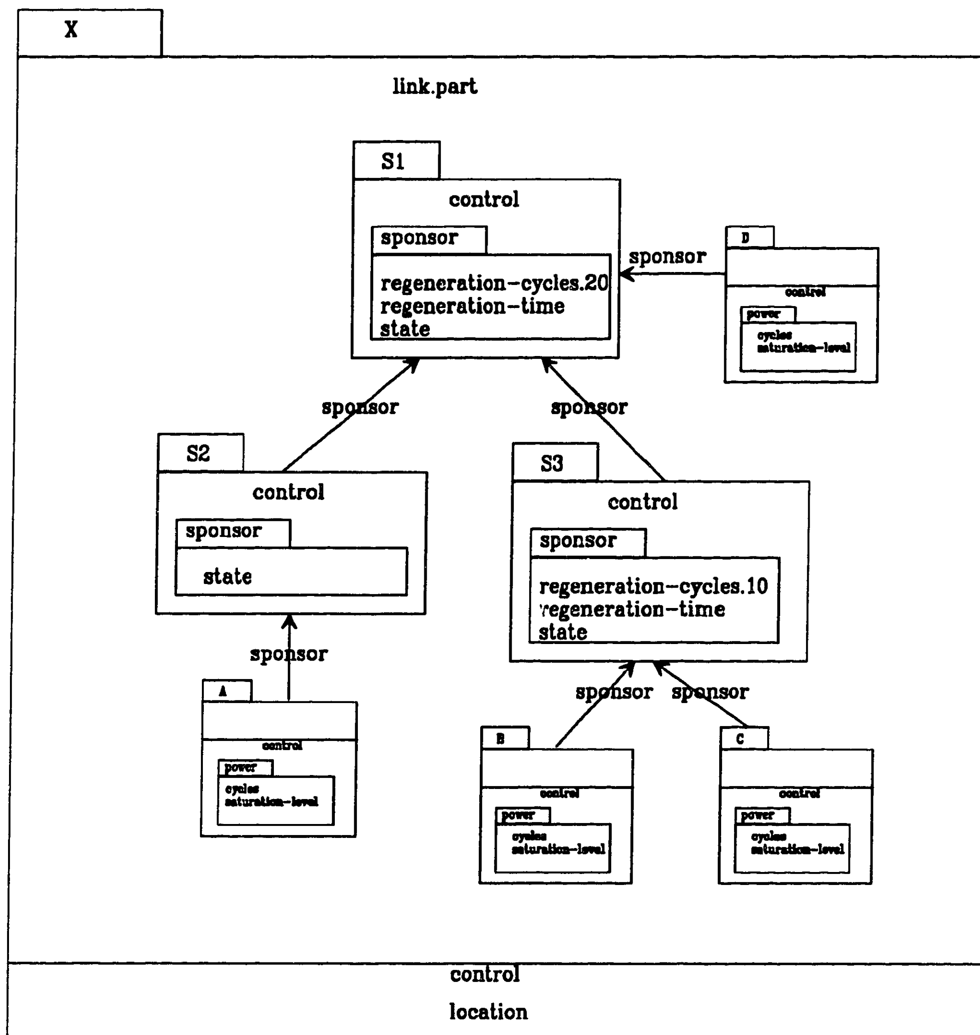


Figure 7.4: Coordinated control is achieved by having multiple regeneration sponsors in one structure.

7.2 Interacting Sponsors

When a configurator with a sponsor is sent to another configurator with a sponsor, the cycles from both sponsors are combined. In figure 7.5, configurator A with sponsor S1 is sent to configurator B with sponsor S3. The processing power of B to process the message A is calculated by combining the cycles in the power command for each configurator. In this example, it would be 15. When the sponsors regenerate the cycles in each configurator, these cycles will also be combined. The saturation level will be the maximum of the saturation level for A or B.

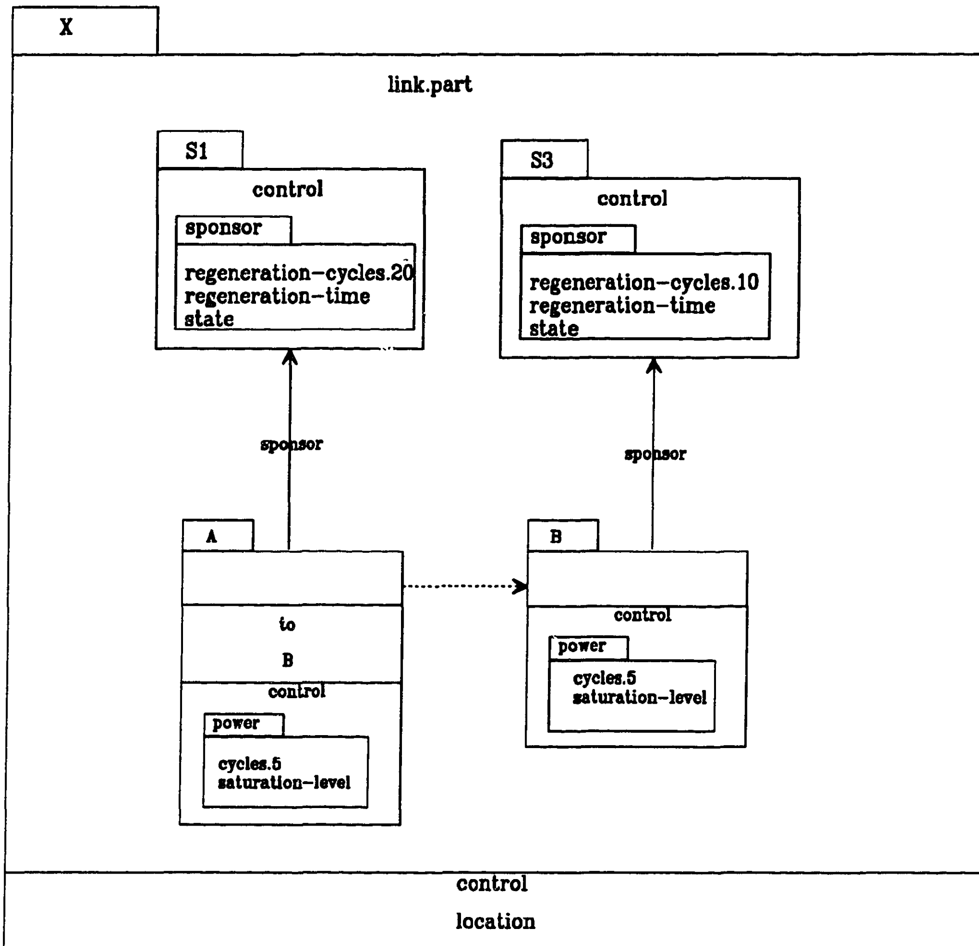


Figure 7.5: Sponsor interaction occurs when a configurator is sent to another as a message. A and B will combine their cycles when B processes message A.

Chapter 8

Development

Organizations continually change: the environment in which they reside changes; new applications are added; old applications are extended. A computer system which represents the organization has to change so that it is still relevant to the organization. The failure to change leads to the obsolescence of the computer system. In traditional computer systems, maintaining a system is costly. In these systems, the computer representation of the organization is quite different from the human mental conception of the organization. The translation between these representations is difficult. If the original system developers are no longer involved with the system, much of the translation knowledge is lost. Ubik's representation of the organization is at a higher, more human, understandable level. In addition, Ubik can read and reason about its representation. This self-reasoning, along with Ubik's control of the action, permits it to measure the mismatch between the computer representation of the organization and the actual organization. Some of the ways this mismatch appears within Ubik follows:

1. The rate and kind of errors generated.
2. The difference in representation between a prototype, and the configurators generated from the prototype.
3. The rate of increase in the number of censors placed on frequently activated procedures.
4. The number of deadlines missed.
5. The number of frequently executed goals which cannot be solved.
6. The amount of organizational deadlock.
7. The rate of increase in suspended messages.
8. The increasing interactive processing in established applications. This indicates that the established application coverage is decreasing.

Development in Ubik occurs in two ways: from an interactive dialogue between Ubik and an end-user, or by Ubik using the mismatch measures in order to reduce the difference between its internal representation and the external organization. Ubik performs the following type of developments:

1. message elimination
2. regrouping
3. reclustering
4. regression
5. prototype development
6. bureaucratic development

Reorganization must take into account the full range of organizational activities. If an organization does not have to react to a changing environment, then the reorganizing will continue until the organizational description becomes *saturated*, that is until the organizational description perfectly match the organizational action. Saturation is a concept described by Stanley Cavell in *Pursuits of Happiness, The Hollywood Comedy of Remarriage* [19] in the context of genre.

> ... a narrative or dramatic genre might be thought of as a medium in the visual arts, or a "form" in music. The idea is that the members of a genre share the inheritance of certain conditions, procedures and subjects and goals of composition, and that in primary art each member of such a genre represents a study of these conditions, something I think of as bearing the responsibility of the inheritance. There is, on this picture, nothing one is tempted to call *the* features of a genre which all its members have in common. First, nothing would count as a feature until an act of criticism defines it as such. (Otherwise it would always have been obvious that, for instance, the subject of remarriage was a feature, indeed a leading feature, of a genre.) Second, if a member of a genre were just an object with features then if it shared *all* its features with its companion members they would presumably be indistinguishable from one another. Third, a genre must be left open to new members, a new bearing of responsibility for its inheritance; hence, in the light of the preceding point, it follows that the new member must bring with it some new feature or features. Fourth, membership in the genre requires that if an instance (apparently) lacks a given feature, it must compensate for it, for example, by showing a further feature "instead of" the one it lacks, Fifth, the test of this compensation is that the new feature introduced by the new member will, in turn, contribute

> to a description of the genre as a whole. But I think one may by now feel that these requirements, thought about in terms of "features," are beginning to contradict one another. ...
>
> Take an example. I have mentioned that one feature of the genre of remarriage will be the narrative's removal of the pair to a place of perspective in which the complications of the plot will achieve what resolution they can. But *It Happened One Night* has no such settled place; instead what happens takes place on the road. I say that what compensates for this lack is in effect the replacement of a past together by a commitment to adventurousness, say to a future together no matter what. But then it will be found that adventurousness in turn plays a role in each of the other films of remarriage. And one may come to think that a state of perspective does not require representation by a place but may also be understood as a matter of directedness, of being on the road, on the way. In that case what is "compensating" for what? Nothing is lacking, every member incorporates any "feature" you can name, in its way. It may be helpful to say that a new member gets its distinction by investigating a particular set of features in a way that makes them, or their relation, more explicit than in its companions. Then as these exercises in explicitness reflect upon one another, looping back and forth among the members, we may say that the genre is striving toward a state of absolute explicitness, of expressive saturation. At that point the genre would have nothing further to generate. This is perhaps what is sometimes called the exhaustion of conventions. There is no way to know that the state of saturation, completeness of expression, has been reached.

8.1 Message Elimination

Message elimination refers to the partial elimination of messages used to run applications within organizations. Eliminating messages reduces the cost of communicating the message and the coordination required to respond to the message. The cost of message elimination is the removal of the control supported by the message. Ubik needs to analyze the message traffic in an organization, eliminate some messages, and substitute tapeworms to establish some of the control lost with the eliminated message.

The purchase organization example from chapter 3 uses messages to initiate and coordinate the actions of multiple departments. The MIT purchase organization system, as described in the *Guide to MIT Administrative Offices* [5], contains two methods of message elimination: blanket purchase orders, and purchase-order drafts.

Blanket purchase orders are pre-approved purchase-orders, as shown in figure 8.1. Purchase-order drafts are purchase-orders in which the payment

goes to the vendor with the purchase-order, as shown in figure 8.2.

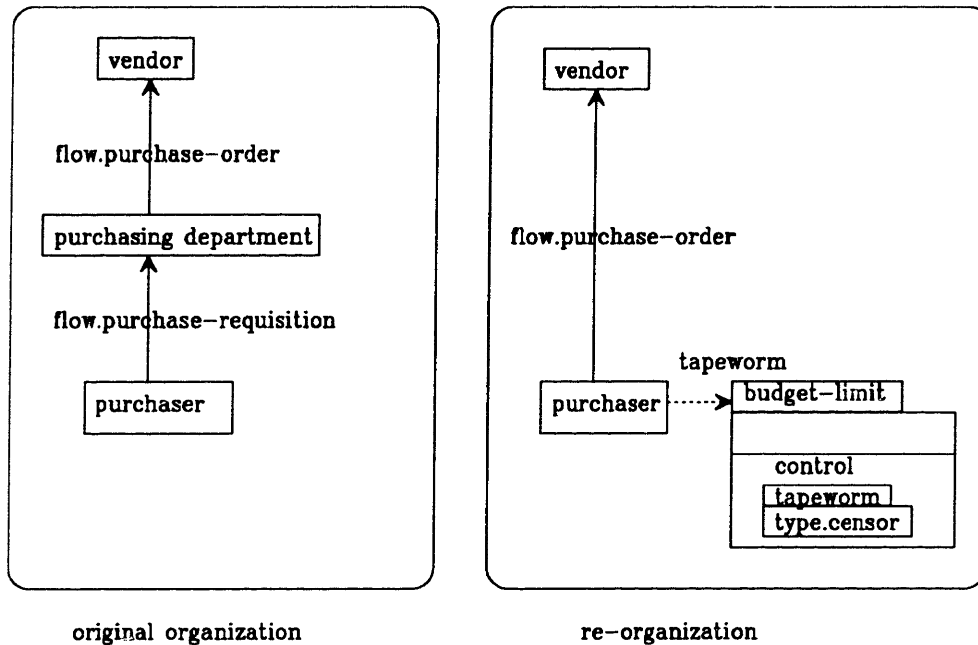


Figure 8.1: Blanket purchase orders are pre-approved purchase-orders. The purchase-requisition goes to the purchasing department for initial approval. Thereafter the purchaser can issue a purchase-order directly to the vendor. The figure on the left is the initial organization; the figure on the right is the organization for the blanket purchase order. Blanket orders save the step of the purchasing department processing the purchase-requisition for each purchase-order. Blanket orders are usually issued with budget limits. The tapeworm provides a censor which maintains the budget limit constraint.

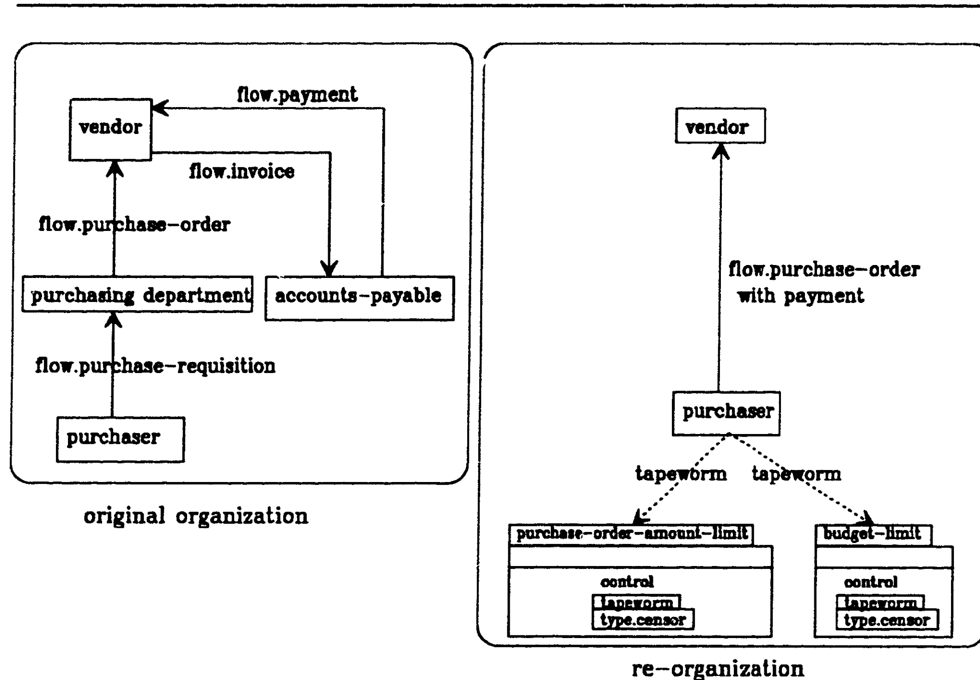


Figure 8.2: Purchase-order drafts are purchase-orders in which the payment goes to the vendor with the purchase-order. It eliminates the need to receive and process the vendor invoice. The figure on the left is the initial organization; the figure on the right is the organization for the purchase-order draft. The purchase-order draft eliminates the step of the purchasing department processing the purchase-requisition for each purchase-order, and the accounts payable department processing the invoice. Purchase-order drafts are usually issued with budget and purchase-order limits. The two tapeworms provide censors to maintain these constraints.

8.2 Regrouping

Regrouping is a method of eliminating messages by batching input messages, and periodically sending the batched input messages in one output message. The batched input messages are usually transformed into the output message, as shown in figure 8.3. In this example, multiple input purchase-orders are transformed into one invoice. The billing configurator in this example works as follows:

1. The input section contains a purchase-order form which specifies that a configurator will be triggered under the following conditions:
   (a) A purchase-order arrives.
   (b) The time is 16:00, as specified by the timer command in the control section.
   (c) The company's bill day is today. The bill day is found by a quester which searches the customer-file; it is bound to variable ?bill-day. The boolean expression compares the bill day to today's date. If they are the same, the billing configurator triggers, and all the customers with today's bill day are processed by the configurator.
2. The action section contains three configurators. The invoice-no configurator produces the invoice number. The date configurator produces today's date. The invoice configurator creates an invoice. The invoice fields are calculated as follows:
   (a) The po-numbers are contained in the variable ?po. The union operation specifies that one invoice is produced for all the purchase-orders.
   (b) The company and date are input variables.
   (c) The item fields sum the totals from all the purchase-orders. The quantity `(+ ?q)` is the sum of all quantities for a part-number ?p, description ?d and unit-price ?u. The amount `(* quantity.? ?u)` is the quantity value multiplied by the unit price.
   (d) The total field `(+ item.amount.?)` is the sum of all the amount fields for all the items.

Regrouping eliminates the communications and processing costs of sending invoices more frequently. On the negative side is the loss of interest as a result of receiving payment at a later time. Electronic funds transfer is changing the cost advantage of regrouping. It drastically reduces the cost of sending a message, as the following *New York Times* article [57] indicates. It will soon be practical to bill the customer for each purchase-order immediately.

Like many corporations, Sears, Roebuck & Co. used to love paying suppliers by mail. As far as Sears executives were concerned, the longer the check took to arrive, the better – all the more interest income for Sears. But these days, the preferred way of paying is electronically transferring money from Sears' bank account to suppliers' accounts. The advantages of electronic payments – lower processing and postage costs, fewer clerical errors and a more predictable cash flow – more than compensate for any loss in interest income, according to companies like Sears, Du Pont and RJR Nabisco...

The cost and complexity of setting up an automated system varies depending on the extent to which the corporation is computerized, but it can be as little as $100,000. "Regardless of how much it costs, it won't take too long to recoup the investment," said Dean A. Bitner, a Sears assistant treasurer. Sears saves about 40 cents for every electronic payment it makes and its suppliers can save up to $1.10, he said. Harp said GE saves between 75 cents and $1.50 for every electronic payment received instead of a check.

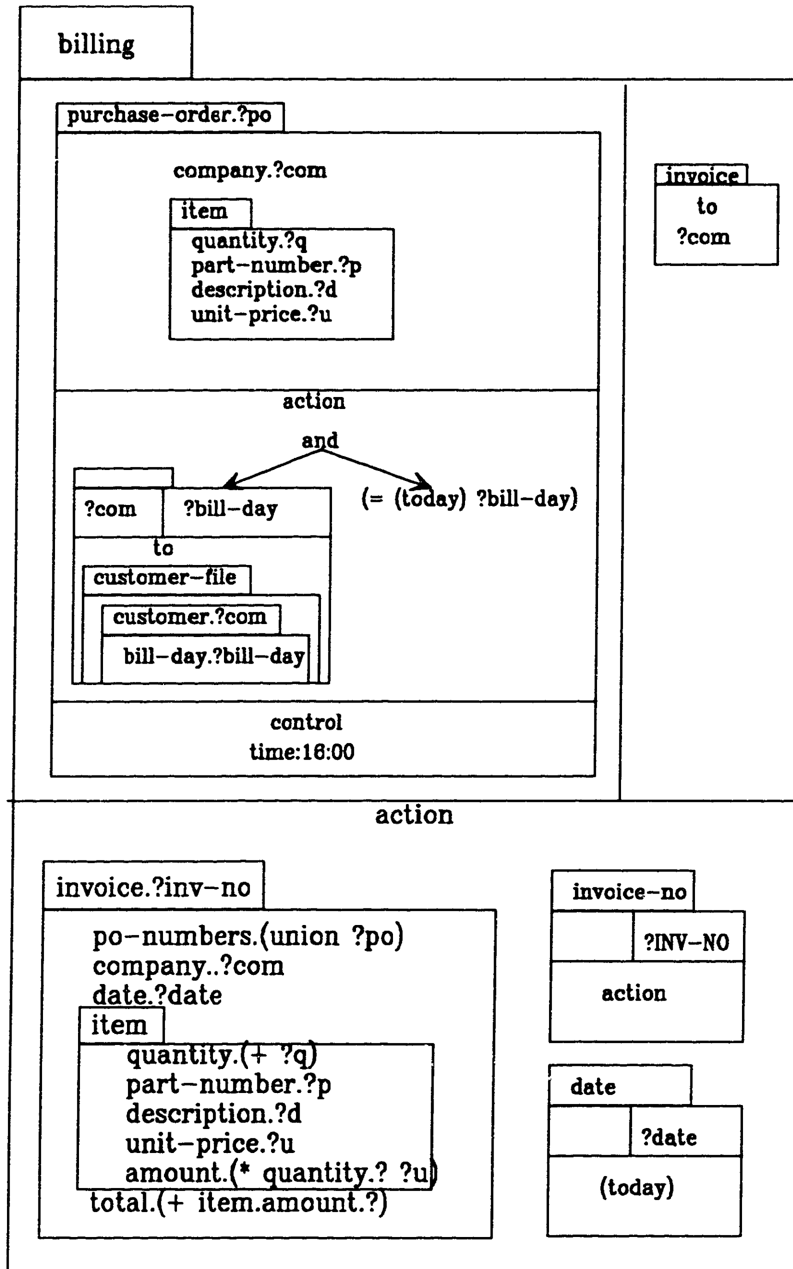


Figure 8.3: The regrouping of purchase-orders into invoices on a monthly basis. The accounts receivable department receives purchase-orders and produces the invoices.

8.3 Reclustering

Links can cross distributed configurator boundaries. Reclustering is a process of reorganization, where the configurators are migrated into the distributed configurator which most frequently references them. Reclustering is illustrated in figure 8.4. The original organization has the employee.jill located in the file.personnel. It has three configurators part linked to it. The salary and mgr configurators are located in file.personnel, and the address configurator is remotely located in file.name-and-address. The reorganization has placed all the part configurators in the same location, which is file.personnel.

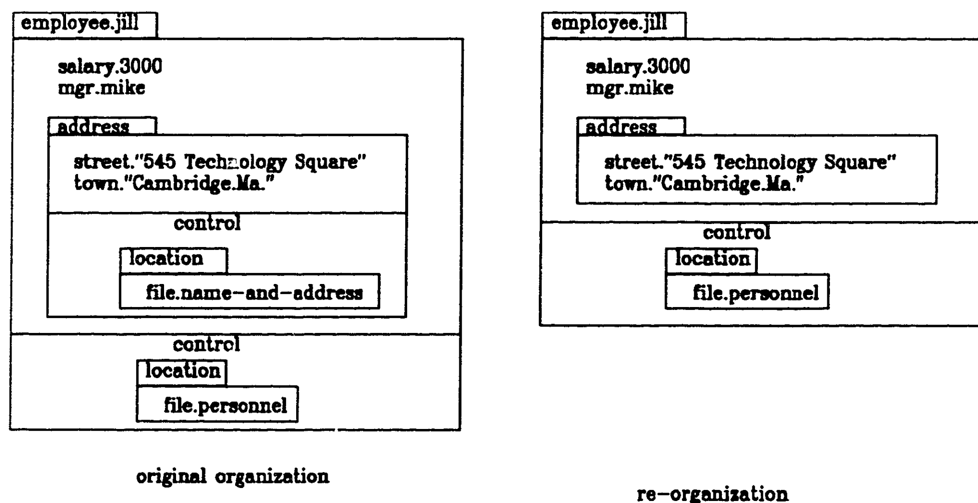


Figure 8.4: Reclustering moves configurators remotely referenced in a distributed configurator into the local distributed configurator. The address part is initially in file.name-and-address. In the re-organization it was moved into the same location as the rest of the employee parts.

8.4 Regression

Regression is the process of moving a tapeworm closer to the source of the activity which it is monitoring or censoring. Regression is illustrated in figure 8.5. In this example, the amount-limit-tapeworm is censoring all the purchase-requisitions received by the purchasing department. Regression can move the tapeworm closer to the location where the purchase-requisitions are being produced. In this example, the tapeworm is moved onto the purchase configurator. The operation is changed from censoring the purchase-requisitions received to censoring the purchase-requisitions sent.

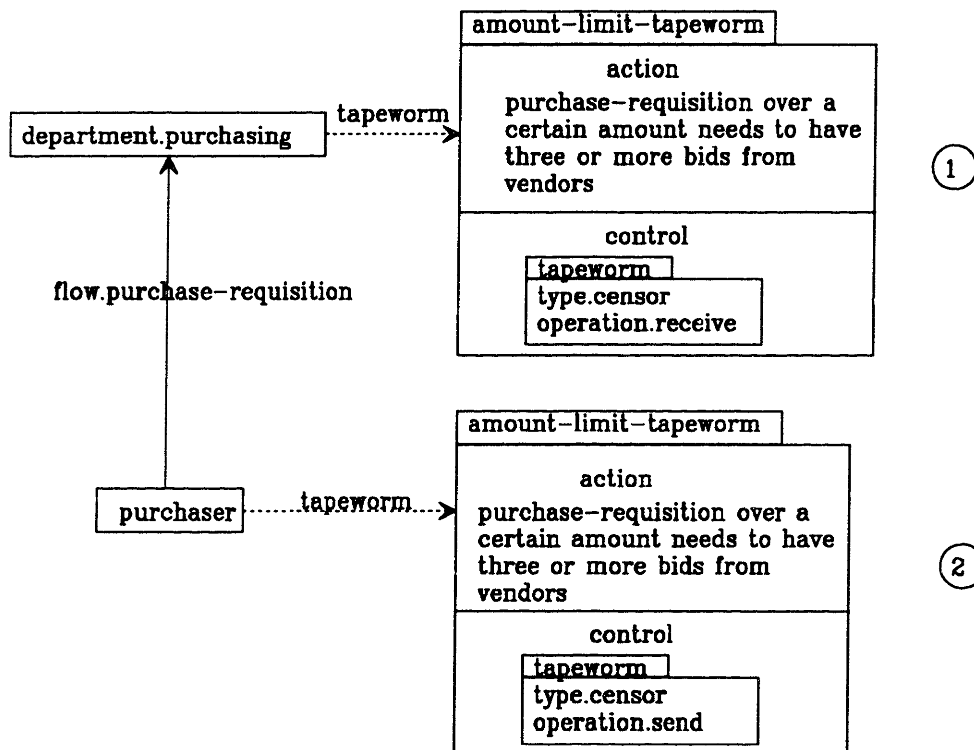


Figure 8.5: Regression of a tapeworm (1) to tapeworm (2). Tapeworm (1) is placed on the department.purchasing to censor all the purchase-requisitions over a certain amount which do not have at least three bids. Regression is used to move the tapeworm closer to the activity in which the bids are obtained. Regression has determined that all purchase-requisitions are issued by an employee with the role purchaser. Tapeworm (2) moves tapeworm (1) onto the role of a purchaser. It changes the operation from receive to send. This is the earliest point at which a completed purchase-requisition can be censored, since before this time the purchase-requisition is not yet complete.

8.5 Prototype Development

A prototype serves as a characterization of a collection of configurators. Configurators are continually created and modified. Over time a prototype might no longer be an adequate characterization of the collection of configurators created from it. Alternately, a configurator might no longer belong in the collection characterized by the prototype. Some of the indications for prototype mismatch are as follows:

1. The inability to find an adequate prototype for use in the creation of a configurator.
2. Significant deviation of individual configurators from their prototype.
3. A configurator is better characterized by a prototype different from its current prototype.

Some configurators are part of a prototype collection for reasons other than structure. For example, a configurator representing a natural kind such as a bird will use a bird as a prototype no matter how its structure changes, unless the concept of a bird undergoes change, or a prototype representing an action or goal has no structure in common with the configurator it represents.

8.5.1 Forming Collections

The first step in developing a prototype is to create a collection of configurators. Some of the methods for creating collections within Ubik follow:

1. **Instances** - The instances of a prototype.
2. **Individuals in all contexts** - Similarly named configurators in different contexts.
3. **Similarity** - Configurators with the same configurator part linked to it.
4. **Reference** - A collection of explicitly referenced configurators.
5. **Action** - All configurators which take place in a common action.

8.5.2 Creating a Prototype from a Collection

Once the collection is formed, the prototype must be created. Below are some ways to create prototypes, the last three of which were described by Tversky [62]:

1. **Designation** - the prototype is specified directly, without regard to any characteristics of the collection.

2. **Similarity** - the prototype is created out of some attributes which occur frequently within the configurators of the collection. The prototype created does not necessarily match any one configurator within the collection, but is *best* in some sense in representing the whole collection.

3. **Exemplar** - the prototype is chosen from one of the configurators within the collection which is determined to best represent the collection.

4. **Family resemblance** - the collection as a whole serves as the prototype. A given configurator is declared as part of the collection if it matches this collection better than another.

8.5.3 Finding a Prototype

Given a configurator, some of the ways of choosing the prototype or collection to which the configurator belongs follow:

1. **Designation** - the collection is specified.

2. **Name** - the name of the configurator determines its prototype.

3. **Metonymic prototype** - the name of the configurator determines its prototype indirectly. For example, the prototype of employee is not necessarily the prototype employee. George Lakoff in *Women, Fire, and Dangerous Things* [46] describes Metonymy as the taking of one well-understood or easy-to-perceive aspect of something and using it to stand either for the thing as a whole or for some other aspect or part of it. A Metonymic prototype of employee might be employee.manager rather than employee.

4. **Similarity** - given a configurator with some attributes, a prototype is chosen which is similar to this configurator. Marvin Minsky in *The Society of Mind* [53] gives a mechanism to accomplish this, which works as follows: the configurators which match the attributes are chosen. If the chosen configurators contain other attributes, the configurators with noncompatible attributes are removed. The remaining configurator with the closest structure to the given configurator is chosen as the prototype.

5. **Metaphoric** - Given a configurator with some attributes, a prototype is chosen by some mapping function. For example, a configurator might be related to a prototype because they both cause a similar batching operation.

8.6 Bureaucratic Development

Bureaucratic development refers the adaptation of an organization to changing application and environmental conditions. Bureaucratic development can be used to expand the purchasing application from a non-shared purchase system, which only supports one purchaser per item, to a shared system, which supports multiple purchases per item. It can also be used to adapt an organization for cooperation with other organizations. Bridging strategies are bureaucratic development techniques to enhance the security of an organization by increasing the number and variety of linkages with competitors. The following is Scott's typology of bridging techniques [59]:

1. **Bargaining** is a family of strategies by means of which the focal organization attempts to ward off dependence. Bargaining is a pre-bridging strategy which is used to establish the other bridging strategies.

2. **Contracting** is the negotiation of an agreement for the exchange of performances in the future. This reduces uncertainty by coordinating future behavior, in limited and specific ways, with other units. In Ubik, contracting permits simplification of internal organizational procedures because of message protocol agreements between organizations.

3. **Cooptation** entails the incorporation of representatives of external groups into the decision-making or advisory structure of an organization. Ubik looks at an organization as a semi-fluid collection of objects which can migrate between suborganizations. Established and maintained organizational boundaries can prevent objects from migrating. Cooptation is a constrained easing of these boundaries.

4. **Joint Ventures** occur when two or more firms create a new organization to pursue some common purpose. This entails only a limited pooling of resources. Ubik's flexible organizational representation would allow existing applications to be easily reorganized for joint ventures.

5. **Mergers** is a strategy in which separate organizations are combined. Like joint ventures, Ubik would allow the maintenance of existing applications during mergers.

6. **Associations** are arrangements which allow similar organizations to work in concert to pursue mutually desired objectives. An example is a trade association or cartel.

7. **Governmental Connections** affect transactions among organizations by helping to determine the overall context of organizational action, placing constraints on selected organizations or activities, and providing resources for organizations.

8. **Institutional linkages** provide concepts, structure, and action which support their existence. For example, a school organization seeks accreditation, which defines how the school is to organize and act.

Chapter 9

Conclusion

Ubik is a system within whose framework the following issues are being explored: high-level distributed language development, organizational representation, the relationship between organizational structure and action, end-user programming, gradual automation, the support of competing and cooperative applications, and the maintenance of organizations consisting of interrelated applications.

Ubik contains an object-oriented language for developing interrelated, distributed, and parallel applications. As a language, Ubik is at the same level as frame and rule based systems; this is higher than procedural languages, and lower than natural languages. Ubik has language and system constructs for distributed and parallel operation. These constructs include tapeworms, questers, constructors, sponsors, distributed configurators, and links which cross computer network boundaries. The unification of configurators traveling as messages with the receiving configurators provides the basic reasoning mechanism within Ubik. This technique allows the organizational action to proceed with partial information. Variables are used to indicate the information which is not yet known. As the action proceeds, these variables are incrementally bound. Much of the organization action takes place in physically distributed locations. Variables in messages provide a means for communicating between the distributed locations, and to perform distributed reasoning.

Ubik is a language for representing organizations, such that there is a close correspondence between the structure and activities of the external organization and the structure and activities of the Ubik representation. Some small scale applications have been represented in Ubik. Further work needs to be done to see if the Ubik representation scales up.

Ubik is a language for describing an organization's structure and action. The structure is represented by a semantic net, consisting of message passing objects. The action consists of messages between the objects. Ubik has produced a tight integration between a message passing system and a semantic net system. The messages can be explicitly passed between objects,

or implicitly passed using flow links. Bureaucratic paths allow an organization to increase its computational power by the incremental introduction of new organizational levels. These levels make new decisions on message flow without requiring changes in existing configurators. Tapeworms can trigger on the sending or receiving of messages. Tapeworm censors can provide replacement behavior for a message passing object.

Ubik is a system for end-user programming. Only a subset of Ubik has been implemented, as discussed in appendix A. This subset does not include high-level end-user interfaces. Query-by-Example, a relational data base system, has proven very successful as an end-user program. Ubik represents an expansion of the interface ideas in Query-by-Example, and should increase the amount of programming an end-user can perform.

The Ubik system supports the gradual automation of a business organization, where the Ubik control of the organization coexists with the manual control and activity of the organization. This support has not been designed or implemented, but the approach would be to replace the action section of a configurator with an end-user interactive dialog handler. When the configurator is triggered, it would interact with an end-user.

Ubik maintains the power relationships between the cooperating and competing applications. A mechanism has been defined to describe power relationships; it has not been implemented. Experiments are needed to find the values of the sponsor and power attributes which would effectively focus the organization's attention.

Ubik supports the automatic development of new Ubik representations to more closely match the continually changing business organization. A start has been made on describing some of the ways an organization can develop. Ubik has described a framework for thinking about this issue. An implementation of the full Ubik system would allow the discovery of effective measures for when development should occur, and to experiment with the development techniques.

A Ubik prototype was built [24,25]. This prototype supports the following configurator sections: input, output, action, and link.part. The type of links supported are the following: batch, part, and tapeworm. Only monitor tapeworms have been implemented. The prototype includes a parallel unification algorithm and supports distributed reasoning using variables. The Ubik prototype and its implementation is described in appendix A.

Appendix A

Early Ubik and Its Implementation

Early Ubik is the predecessor system to Ubik, as described in this thesis. The Early Ubik configurator has three sections:

1. **Name** - the name of the configurator. If the configurator resides in a network, the name contains its path through the network. For example, a configurator with a path name `a.b.c` specifies that configurator `c` can be reached by the path `a.b.c`.

2. **Input section** - the input section specifies the message that the configurator will accept.

3. **Output section** - the output section specifies the final message which will be sent or returned by the configurator.

4. **Body section** - the body section is a combination of the Ubik action and link.part section.

The types of links supported in Early Ubik are batch, part, and tapeworm. Not supported are configurator sections `to` and control, tapeworms of type censor, distributed configurators, and links of type flow, label, prototype, sponsor, and value. A prototype was implemented on the Acore parallel processing system [51].

The configurator format is shown in figure A.1.

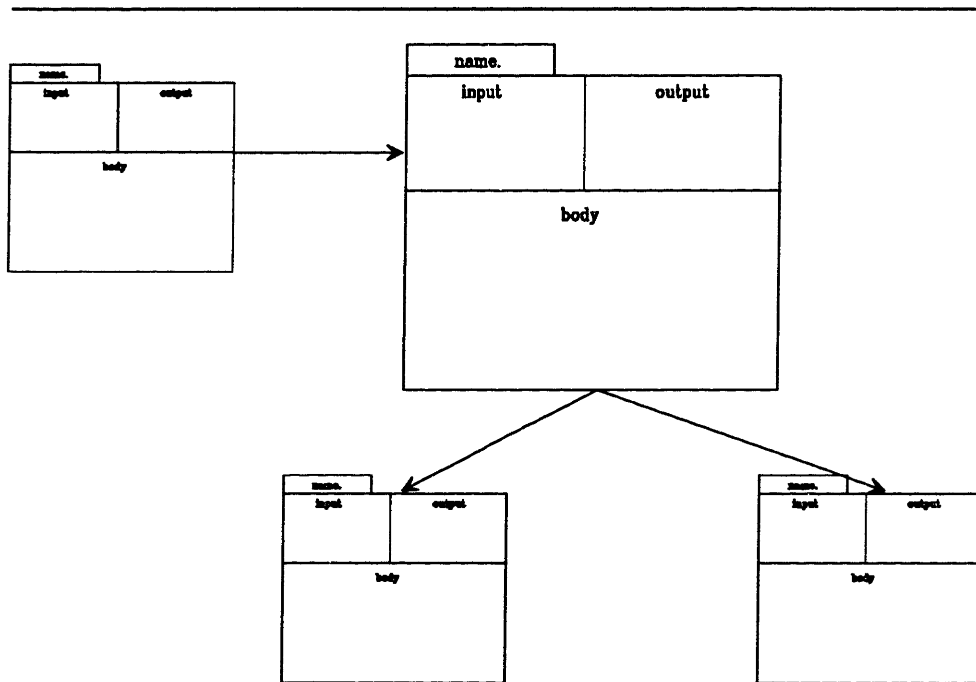


Figure A.1: Early Ubik two-dimensional box notation. The box has a name, input, output, and body section.

The BNF syntax of Early Ubik is shown below. It uses the following metasyntax: [ ] specifies optional clauses; <> specifies non-terminal clauses; * specifies multiple occurrence. | specifies one of the listed clauses.

```
<configurator>=(<name>[<input>][<output>][<body>])

<name>=<atom> |(<atom>*)
              |<atom>.<atom>*
              |(<atom>.<atom>*)

<input>=(INPUT  [<expression>]
                [BATCH <expression>])

<output>=(OUTPUT <expression> [(TO <configurator>*)]
                              [(REPLY expression)]

<body>=(BODY <expression>)|(<expression>)

<expression>=(AND <expression>)|(OR <expression>)
            |(NOT <expression>)|(<WHEN> <expression>)
            |(EVAL-LISP <expression>)|<configurator>
            |(<configurator>*)

<when>=WHEN-INSERTED|WHEN-DELETED|WHEN-UPDATED
      |WHEN-MODIFIED|WHEN-REFERENCED

<list>=(<atom>*)

<atom>=variable|constant
```

Configurators can be assembled into structures to represent an organization. Figure A.2 shows a structure consisting of ten configurators:

This example, in linear syntax, would be as follows, where the dot notation is used to indicate a new network level:

```
(ubik
 (.employee
   (.peter
     (body (salary.30)
           (mgr.mike)))
   (.jill
     (body (salary.40)
           (mgr.mike))))
 (.department
   (.purchasing.buyer.(peter
                        (body (responsibility.computers))))
   (.shipping.clerk.jill)))
```

The body section need not be labeled. Jill in the above example could be written as:

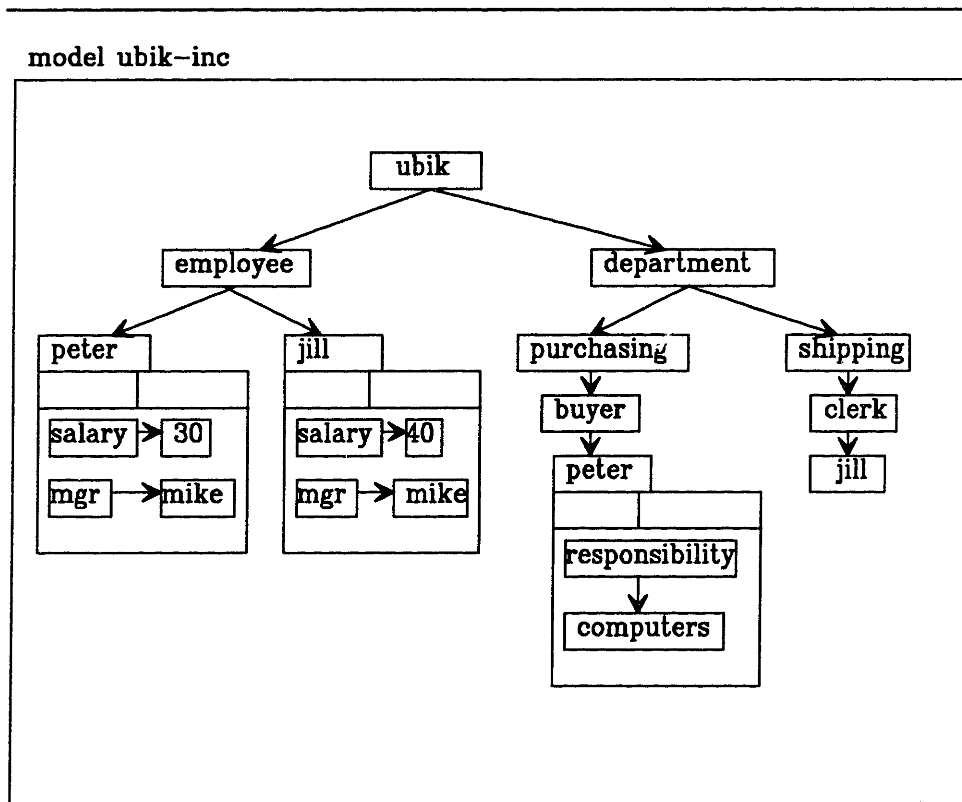


Figure A.2: Organization network consisting of two employees and two departments.

```
(jill
  (salary.40)
  (mgr.mike))
```

A.1 Operations

Early Ubik operations are used to create, modify, and query the configurators.

Early Ubik runs on one processor. Ubik configurators reside in a structure called Ubik memory, which is divided into models. A model consists of a network of configurators. Each model has associated with it a current context. This is a default path used to reference configurators within the model. The following operations are used to create and modify the memory and models:

- (**@reset-memory**) - initializes Ubik memory.
- (**@create-model** name) - creates a new model in memory with the specified name.
- (**@set-current-model** name) - sets the context for the memory to the specified model name.
- (**@set-current-context** location) - sets the context to the specified location.
- (**@expunge-model**) - removes deleted configurators from the networks within a model.

Networks of configurators are built with the following operations:

- (**@I** network) - create or insert the specified network.
- (**@U** network) - update the specified network.
- (**@D** network) - delete the specified network.

Networks are queried using the following operation:

- (**@Q** network) - query the specified network.

Actions are evaluated using the following operation:

- (**@X** configurator) - evaluate the specified configurator.

A.2 Configurator Unification

Unification is the process of matching two configurator patterns and binding any variables within the patterns. The following examples illustrate how unification works within Early Ubik. In each example the first pattern is unified with the second. The resulting variable bindings are then shown.

```
pattern 1 (a.b)
pattern 2 (a.?x)

unify 1 to 2 ((?x=b))

pattern 1 (a (?x))
pattern 2 (a (c)(d))

unify 1 to 2 ((?x=c)(?x=d))

pattern 1 (a.?y (?x))
pattern 2 (a.employee (customer))

unify 1 to 2 ((?y=employee ?x=customer))

pattern 1(a.b.c (?r.e)(?x.e))
pattern 2 (a.b.c (m.e)(l.e))

unify 1 to 2 ((?r=m ?x=l)(?r=l ?x=m))

pattern 1 (a.b.c (?r.e)(?y))
pattern 2 (a.b.c (m.e1)(l.e))

unify 1 to 2 ((?r=l ?y=m))

pattern 1 (a.b.c (?r.e1)(?x.e))
pattern 2 (a.b.c (m.e1)(l.e))

unify 1 to 2 ((?r=m ?x=l))
```

The direction of unification is significant. In the following, pattern 1 will completely match pattern 2, but pattern 2 will not match pattern 1 because configurator (e) does not appear in pattern 1.

```
pattern 1 (a.b (c)(d))
pattern 2 (a.b (c)(d)(e))

unify 1 to 2 - nil
unify 2 to 1 - fail
```

Unification occurs in an environment. In the following example, pattern 3 will unify pattern 4, producing the environment (?x=e1 ?y=b1). Unification of pattern 1 to pattern 2 in this environment will produce a failure because ?y will be bound to d1, which is incompatible with the current binding to ?y of b1.

```
pattern 1 (a.b.c (c1.e1)(a1.?y))
pattern 2 (a.b.c (a1.d1)(x1.?y)(c1.?x1))
pattern 3 (a.b.c (c1.e1)(a1.?y))
pattern 4 (a.b.c (a1.b1)(x1.?y)(c1.?x1))

unify 3 to 4 ((?x1=e1 ?y=b1))
unify 1 to 2 failure in environment ((?x1=e1 ?y=b1))
```

Configurator variables are variables which unify with the configurator rather than the name of a configurator. They are specified by a variable with a double question mark. ??x designates configurator variable x. The following example shows the distinction between the use of a variable and a configurator variable.

```
pattern 1 (a.b.c (?b))
pattern 2 (a.b.c (??b))
pattern 3 (a.b.c (a1.b1)(x1.?y2)(c1.?x2)))

unify 1 to 3 ((?b=a1) (?b=x1) (?b=c1))
unify 2 to 3 ((??b=((a1.b1)(x1.?y2)(c1.?x2))))
```

The bindings to a configurator variable will unify with the bindings of another configurator variable, as illustrated below:

```
pattern 1 (??a)
pattern 2 (??b)
pattern 3 (a.b.c (??a))
pattern 4 (a.?z.c (a1.b1)(x1.?y)(c1.b3))
pattern 5 (a.b3.c (??b))
pattern 6 (a.?w.c (a1.b1)(x1.b2)(c1.?x))

unify 5 to 6 ((?w=b3 ??b=((a1.b1)(x1.b2)(c1.?x))))
unify 3 to 4 ((?z=b ??a=(a1.b1)(x1.?y)(c1.b3))))
unify 1 to 2 ((?w=b3 ?z=b ?y=b2 ?x=b3))
```

A.3 Application Examples

The following examples of Early Ubik were used as regression test cases for the prototype. They illustrate how business applications can be built using Ubik.

A.3.1 Building a Network

The following example builds the network partially illustrated in figure A.2. The first operations set up the two models within the memory: mit and ubik-inc.

```
(@reset-memory)
(@create-model 'mit)
(@create-model 'ubik-inc)
(@set-current-model 'ubik-inc)
```

The following operations build the network:

```
(@I (ubik.employee)))
(@I (ubik.department)))
(@set-current-context (ubik.employee)))
(@I (.peter (salary.30)(mgr.mike))))
(@I (.jill (salary.40)(mgr.mike))))
(@I (.mike (salary.50))))
(@set-current-context (ubik.department)))
(@I (.purchasing)))
(@I (.shipping)))
(@set-current-context (ubik.department.purchasing)))
(@I (.buyer.peter (responsibility.computers))))
(@set-current-context (ubik.department.shipping)))
(@I (.clerk.jill (lifting-weight.150))))
```

The following operations modify the network. Jill's lifting weight is changed from 150 to 125, and the employee peter is deleted and expunged.

```
(@set-current-context (ubik.department.shipping)))
(@U (.clerk.jill (lifting-weight.125))
(@set-current-context (ubik.employee)))
(@D (.peter))
(@expunge-model))
```

A.3.2 Message Sending and Receiving

In the following example an order is sent to the accounting department and an invoice is returned. Figure A.3 graphically illustrates this example.

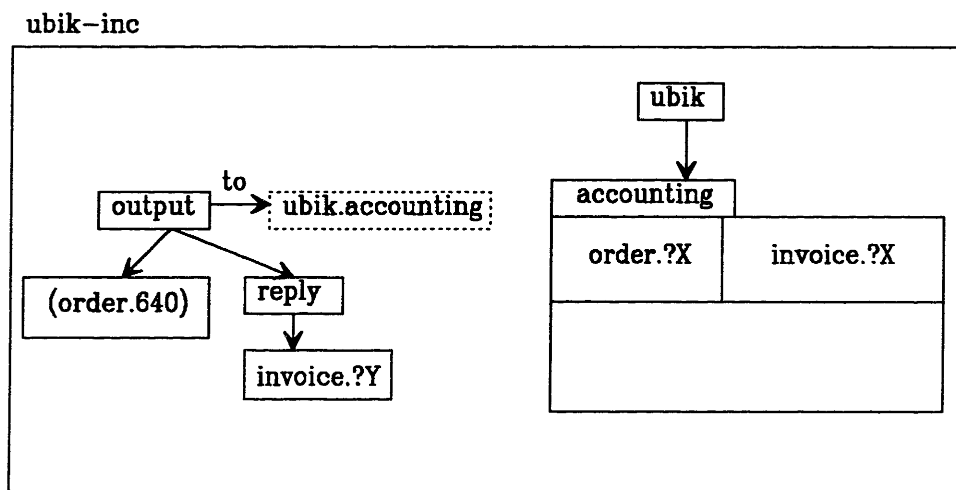


Figure A.3: Message sending in which an order is sent to the ubik accounting department.

1. Build an accounting configurator which would receive orders and send invoices.

```
(@I (ubik.accounting
      (input (order.?X))
      (output (invoice.?X)))))
```

2. Send an order message to the accounting department and receive an invoice in reply.

```
(@X (output (order.640)
             (to (ubik.accounting))
             (reply (invoice.?Y)))))
```

The order message will unify with the input pattern as follows:

```
message (order.640)
input   (order.?x)

unify message to input ((?x=640))
```

The reply pattern will unify with the output message as follows:

```
reply  (invoice.?y)
output (invoice.?x)
environment ((?x=640))

unify reply to output ((?y=640))
```

Instantiation replaces variables with their values.

```
reply       (invoice.?y)
environment (?x=640)

instantiate reply (invoice.640)
```

A.3.3 Unifying with the Configurator Body

When a message is sent to a configurator the following occurs:

1. The message is unified with the input section.

2. The input section is unified with the body, if the body contains a network of linked configurators.

3. The output message specified in the output section is sent.

In the following example an invoice message is received by the accounting configurator and unified with its body, as graphically illustrated in figure A.4.

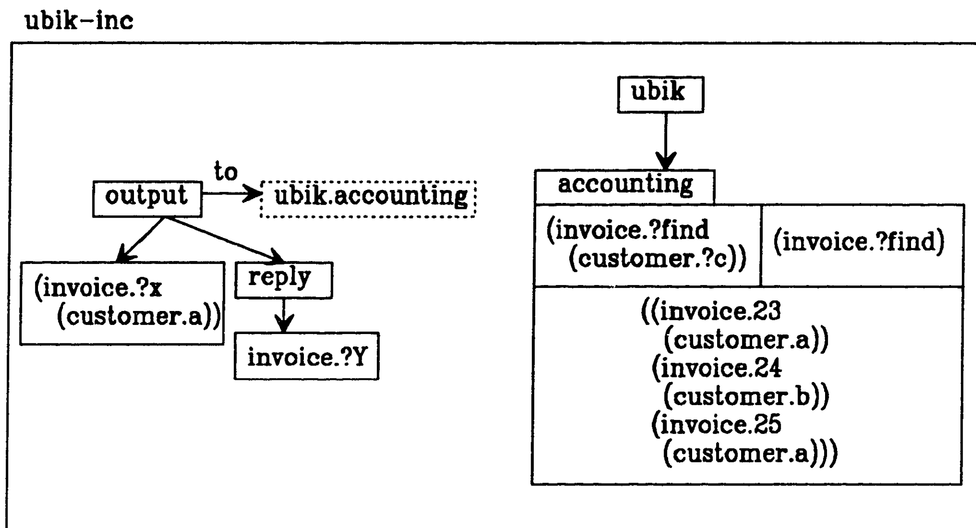


Figure A.4: Unifying with the configurator body occurs when a message is sent to a configurator which has a network of linked configurators in its body.

1. Build accounting configurator with an input and output pattern and a body of linked parts.

```
(@I (ubik.accounting
      (input (invoice.?find
               (customer.?c)))
      (output (invoice.?find))
      ((invoice.23
         (customer.a))
       (invoice.24
         (customer.b))
       (invoice.25
         (customer.a)))))
```

2. Send an invoice message to the accounting configurator.

```
(@X (output (invoice.?x
              (customer.a))
            (to (ubik.accounting))
            (reply (invoice.?Y)))))
```

The unification will be as follows:

```
input   (invoice.?find (customer.?c)))

output (invoice.?find)

body    ((invoice.23 (customer.a))
          (invoice.24 (customer.b))
          (invoice.25 (customer.a))))

message (invoice.?x (customer.a))

reply    (invoice.?Y)


unify message to input ((?x=?find ?c=a))

unify input to body     ((?find=23) (?find=25))

unify reply to output   ((?y=?find))

instantiate reply ((invoice.23)(invoice.25))
```

If the output box is empty, then the input section message will also become the output section message. In the example below, the output box is empty and the invoice has a body which contains both a customer and salesman. This example will return the same result as above.

```
(@I (ubik.accounting
     (input (invoice.?find
              (customer.?c)))
     (body
       (invoice.23
         (customer.a)
         (salesman.joe))
       (invoice.24
         (customer.b)
         (salesman.mike))
       (invoice.25
         (customer.a)
         (salesman.jill)))))

(@X (output (invoice.?x
              (customer.a))
              (to (ubik.accounting)))))
```

A.3.4 Configurator Variables

Configurator variables can be used in the input section to allow the configurator to accept a variety of messages, as illustrated in figure A.5. In this example any message received is accepted and unified with the configurators in the body.

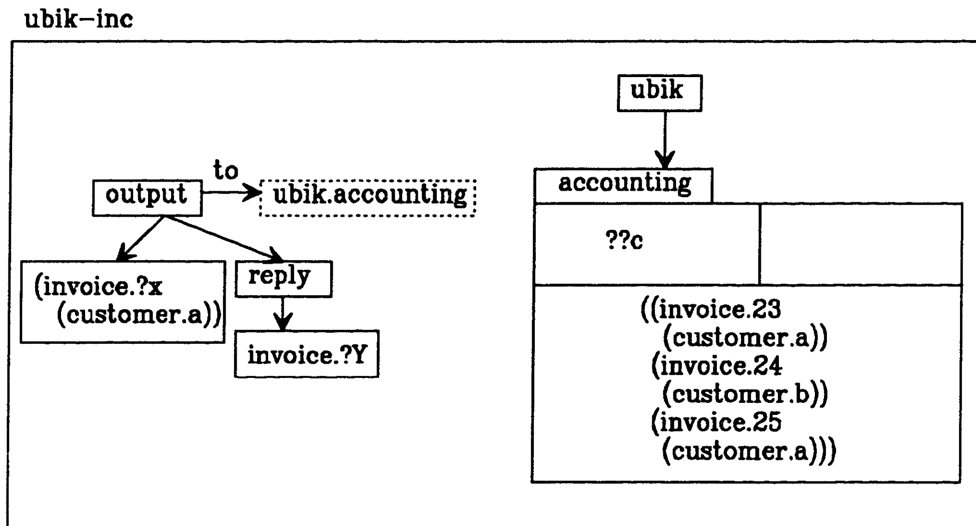


Figure A.5: A configurator variable binds to the input configurator. In this example the configurator which binds to ??c will be directly unified with the configurators linked to the body of the configurator.

A configurator variable can be imbedded in a configurator. In the example below the accounting configurator will accept all invoice messages. The configurator variable ??c will bind to the body of the invoice message.

```
(@I (ubik.accounting
      (input (invoice.?y
                (??c)))
      ((invoice.23
         (customer.b)
         (salesman.joe))
       (invoice.24
         (customer.b)
         (salesman.mike))
       (invoice.25
         (customer.a)
         (salesman.mike))
       (invoice.26
         (customer.1)
         (salesman.jill)))))
```

Below are messages sent to the accounting configurator and the replies returned.

1. Invoices with customer.b.

```
(@X (output (invoice.?x
             (customer.b))
            (to (ubik.accounting)))))
```

The reply is:

```
((invoice.23 (customer.b)(salesman.joe))
 (invoice.24 (customer.b)(salesman.mike)))
```

2. Invoices with salesman.mike.

```
(@X (output (invoice.?x
             (salesman.mike))
            (to (ubik.accounting)))))
```

The reply is:

```
((invoice.24 (customer.b)(salesman.mike))
 (invoice.25 (customer.a)(salesman.mike)))
```

3. All invoices in ubik.accounting.

```
(@X (output (invoice.?x
             (??m))
            (to (ubik.accounting)))))
```

The reply is:

```
((invoice.23 (customer.b)(salesman.joe))
 (invoice.24 (customer.b)(salesman.mike))
 (invoice.25 (customer.a)(salesman.mike))
 (invoice.26 (customer.l)(salesman.jill)))
```

4. Invoices with salesman.mike and customer.b.

```
(@X (output (invoice.?x
             (salesman.mike)
             (customer.b))
            (to (ubik.accounting)))))
```

The reply is:

```
((invoice.24 (customer.b)(salesman.mike)))
```

5. Salesman and customer for invoice.24.

```
(@X (output (invoice.24
             (salesman.?s)
             (customer.?c))
            (to (ubik.accounting)))))
```

The reply is:

```
((invoice.24 (customer.b)(salesman.mike)))
```

6. Invoice.24's body.

```
(@X (output (invoice.24
              (??m)
            (to (ubik.accounting)))))
```

The reply is:

```
((invoice.24 (customer.b)(salesman.mike)))
```

7. Salesman and customer for order.24.

```
(@X (output (order.24
              (salesman.?s)
              (customer.?c))
            (to (ubik.accounting)))))
```

The reply is `nil`.

8. Query ubik.accounting for the invoice which contain salesman.mike and customer.b.

```
(@Q (and (ubik.accounting
           (invoice.?x
             (salesman.mike)))
         (ubik.accounting
           (invoice.?x
             (customer.b)))))
```

The reply is:

```
(and (ubik.accounting
        (invoice.24
          (salesman.mike)))
     (ubik.accounting
        (invoice.24
          (customer.b)))))
```

9. Send a message to find invoices which contain salesman.mike and customer.b.

```
(and (@X (output (invoice.?x
                    (salesman.mike))
                 (to (ubik.accounting))))
     (@X (output (invoice.?x
                    (customer.b))
                 (to (ubik.accounting))))))
```

The reply is:

```
(and (invoice.24
       (salesman.mike))
     (invoice.24
       (customer.b)))
```

10. Find the invoices which contain salesman.mike and return its customer.

```
(and (@X (output (invoice.?x
                   (salesman.mike))
                 (to (ubik.accounting))))
     (@X (output (invoice.?x
                   (customer.?y))
                 (to (ubik.accounting))))))
```

The reply is:

```
((and (invoice.24
        (salesman.mike))
      (invoice.24
        (customer.b)))

 (and (invoice.25
        (salesman.mike))
      (invoice.25
        (customer.a))))
```

11. Return the body of the invoices which contains salesman.mike.

```
(and (@X (output (invoice.?x
                   (salesman.mike))
                 (to (ubik.accounting))))
     (@X (output (invoice.?x
                   (??m))
                 (to (ubik.accounting))))))
```

The reply is:

```
((and (invoice.24
        (salesman.mike))
      (invoice.24
        (customer.b)
        (salesman.mike)))

 (and (invoice.24
        (salesman.mike))
      (invoice.25
        (customer.a)
        (salesman.mike))))
```

A.3.5 Distributed Message Send

Messages can be used to collect information from multiple configurators, as illustrated in figure A.6. In this example a message is sent to the invoice-search configurator. The body of this configurator contains an action consisting of an *and expression*. This expression sends two messages in parallel. One message goes to the salesman configurator and the other to the customer configurator. The flow is illustrated in figure A.7.

The Ubik code for this example follows:

1. The salesman configurator is built.

```
(@I (ubik.accounting.invoice.salesman
       (input (invoice.?y
                (??c)))
       ((invoice.23
          (salesman.joe))
        (invoice.24
          (salesman.mike))
        (invoice.25
          (salesman.mike))
        (invoice.26
          (salesman.jill))))))
```

2. The customer configurator is built.

```
(@I (ubik.accounting.invoice.customer
       (input (invoice.?y
                (??c)))
       ((invoice.23
          (customer.b))
        (invoice.24
          (customer.b))
        (invoice.25
          (customer.a))
        (invoice.26
          (customer.1))))))
```

3. The invoice-search configurator is built. This configurator contains an *and expression* in its body which sends messages to the salesman and customer configurators, requesting invoices as specified in the input message.

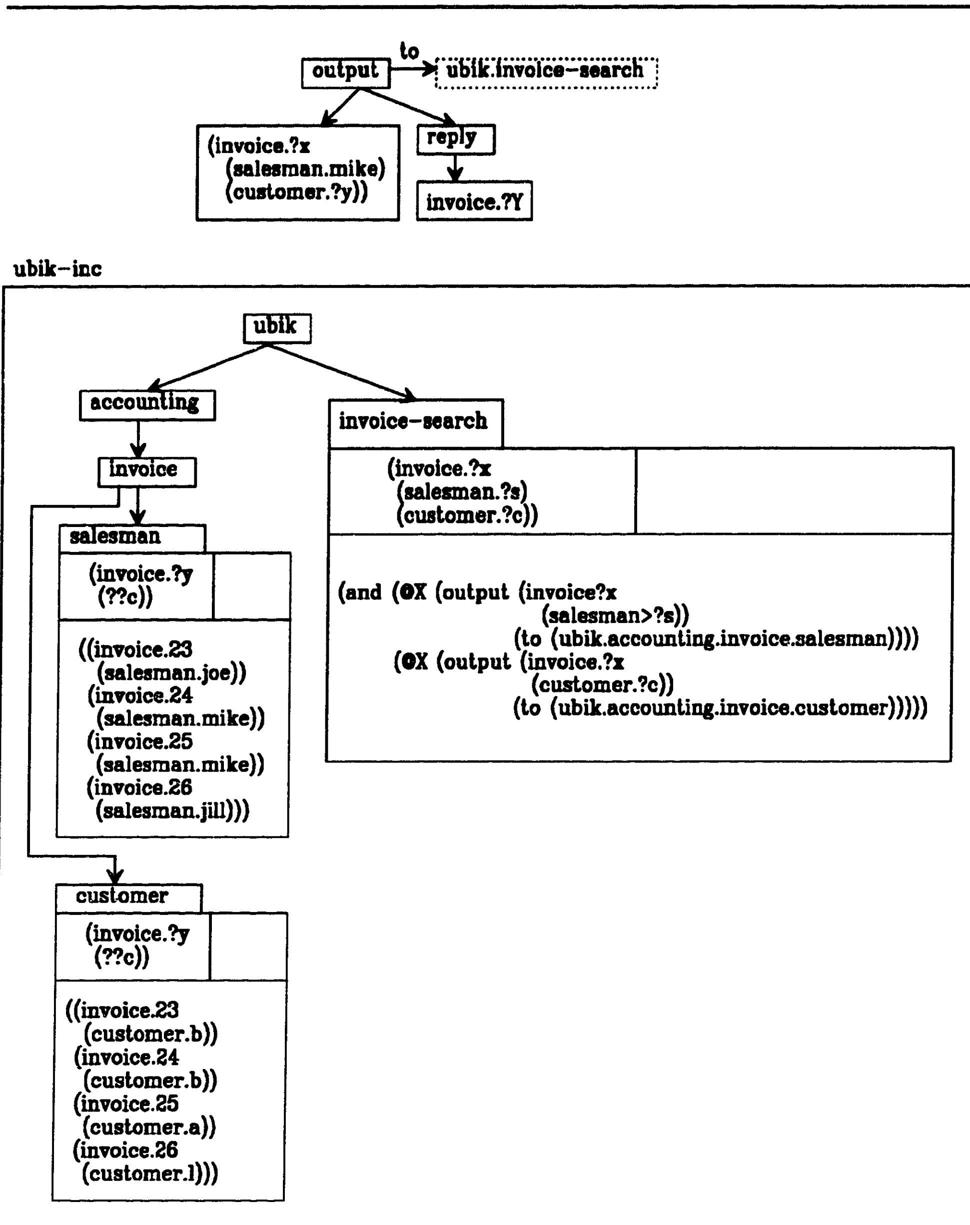


Figure A.6: Parallel *and expression* will process its operands in parallel.

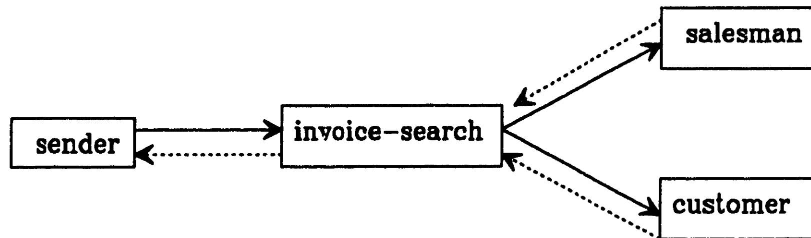


Figure A.7: Parallel *and expression* flow.

```
(@I (ubik.invoice-search
      (input (invoice.?x
               (salesman.?s)
               (customer.?c)))
      (body
        (and
          (@X
            (output
              (invoice.?x
                (salesman.?s))
              (to (ubik.accounting.invoice.salesman))))
          (@X
            (output
              (invoice.?x
                (customer.?c))
              (to (ubik.accounting.invoice.customer)))
                )))))
```

4. This output message requests all the invoices and customers for salesman mike.

```
(@X (output (invoice.?x
              (salesman.mike)
              (customer.?y))
            (to (ubik.invoice-search)))))
```

The reply is:

```
((invoice.24
   (salesman.mike)
   (customer.b))

 (invoice.25
   (salesman.mike)
   (customer.a)))
```

5. A new invoice-search configurator is substituted for the previous one. Its body contains an *or expression* instead of an *and expression*. It

will find all the invoices which have either the specified salesman or the specified customer.

```
(@U (ubik.invoice-search
      (input (invoice.?x
               (salesman.?s)
               (customer.?c)))
      (body
        (or
          (@X
            (output
              (invoice.?x
                (salesman.?s))
              (to (ubik.accounting.invoice.salesman))))
          (@X
            (output
              (invoice.?x
                (customer.?c))
              (to (ubik.accounting.invoice.customer))))
                )))))
```

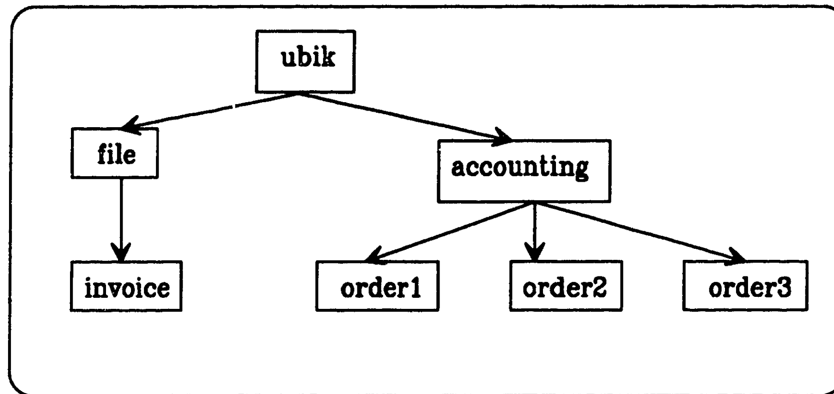
A.3.6 Data Flow

A configurator can be viewed as a filter. The combination of broadcasted messages and configurator filters allows the specification of applications in a data flow style, as shown in figure A.8. In this example a sender sends a message to the configurators order1, order2, and order3. Order1 and order3 accept the message and pass it on to the invoice configurator.

The Ubik code is as follows:

1. The invoice file is built.

```
(@I (ubik.file.invoice
      (input  (??in))
      (output (??in))
      (body (invoice.23
               (salesman.joe)
               (customer.b)
               (item.dress))
            (invoice.24
               (salesman.mike)
               (customer.b)
               (item.auto))
            (invoice.25
               (salesman.mike)
               (customer.b)
               (item.dress))
            (invoice.26
               (salesman.jill)
               (customer.a)
               (item.dress))))))
```



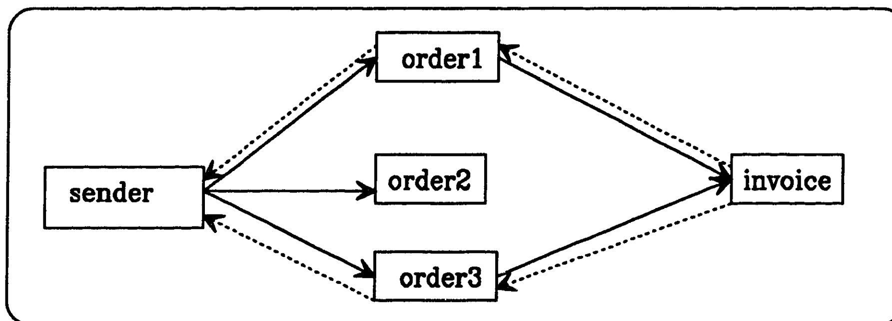


Figure A.8: Data flow control can be achieved by sending messages to a collection of configurators. Each configurator filters the messages it will accept. The top diagram is the organization structure, and the bottom diagram is the message flow.

2. The order1 configurator accepts only order messages with customer.b and outputs invoices with salesman.mike.

```
(@I (ubik.accounting.order1
      (input  (order.?o
                (customer.b)
                (item.?d)))
      (output (invoice.?iv
                (salesman.mike)
                (item.?d))
              (to ubik.file.invoice)))))
```

3. The order2 configurator accepts only order messages with customer.a and outputs invoices with salesman.jill.

```
(@I (ubik.accounting.order2
      (input  (order.?o
                (customer.a)
                (item.?d)))
      (output (invoice.?iv
                (salesman.jill)
                (item.?d))
              (to ubik.file.invoice)))))
```

4. The order3 configurator accepts only order messages with customer.b and outputs invoices with salesman.joe.

```
(@I (ubik.accounting.order3
      (input  (order.?o
                (customer.b)
                (item.?d)))
      (output (invoice.?iv
                (salesman.joe)
                (item.?d))
              (to ubik.file.invoice)))))
```

5. An order message is sent to find invoices with item.dress and customer.b. It will be accepted by configurators order1 and order3 and sent to the invoice configurator. A reply containing the invoice numbers, salesman, and item are expected.

```
(@X (output (order.640
              (item.dress)
              (customer.b))
            (to (ubik.accounting.??))
            (reply (invoice.?I
                     (salesman.?s)
                     (item.?d))))))
```

The reply is:

```
((invoice.23
   (salesman.joe)
   (customer.b)
   (item.dress))

 (invoice.25
   (salesman.mike)
   (customer.b)
   (item.dress)))
```

A.3.7 More than Manager Query

The employee who makes more than his manager query is written as follows:

1. The employee configurator is built.

```
(@I (ubik.file.employee
      (input  (??in))
      (output (??in))
      (body (employee.joe
               (salary.30)
               (mgr.mike))
            (employee.mike
               (salary.20)
               (mgr.bill))
            (employee.bill
               (salary.100))))))
```

2. Two messages are sent to the employee configurator. One will request all the employees with their salary and manager. The other will request all the employees and their salary. The combination of the *and expression* and the > expression will restrict the replies to only the employees who make more than their managers. This example is the first use of a configurator without an input statement. Evaluating the configurator will cause the immediate evaluation of its body. Eval-lisp is a Ubik expression which will invoke the underlying Lisp interpreter for specified expression. Before Lisp is invoked, the variables are instantiated.

```
(@X (ubik.make-more
      (output (employee.?e
                (salary.?s)
                (mgr.?m)))
      (body (and (@X (output (employee.?e
                                (salary.?s)
                                (mgr.?m))
                              (to (ubik.file.employee))))
                 (@X (output (employee.?m
                                (salary.?ms))
                              (to (ubik.file.employee))))
                 (eval-lisp (> ?s ?ms)))))))
```

The reply is:

```
(employee.joe
  (salary.30)
  (mgr.mike))
```

A.3.8 Batching

Batching of input messages is achieved with an *and expression* within the input section, as shown in figure A.9. In this example receiving-tickets and notification-of-receipt messages are received. A batch exists which contains all messages without a matching purchase-order.

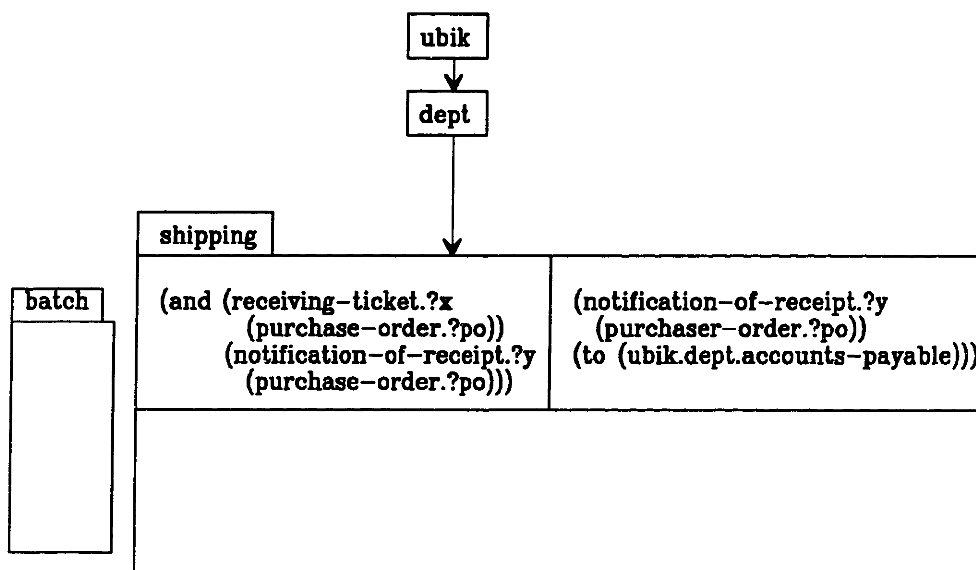


Figure A.9: Batching input messages is achieved with an *and expression* in the configurator input section.

1. The shipping configurator is built which receives and batches receiving-tickets and notification-of-receipts. It sends notification-of-receipts to the accounts-payable configurator when a purchase-order match has occurred.

```
(@I (ubik.dept.shipping
      (input (and (receiving-ticket.?X
                    (purchase-order.?po))
                  (notification-of-receipt.?Y
                    (purchase-order.?po))))
      (output (notification-of-receipt.?Y
                 (purchase-order.?po))
              (to (ubik.dept.accounts-payable)))))
```

2. The accounts-payable configurator receives a notification-of-receipt and places it in the file.payable.purchase-order.

```
(@I (ubik.dept.accounts-payable
      (input (notification-of-receipt.?n
               (purchase-order.?po)))
      (body (@I (file.payable.purchase-order
                   (purchase-order.?po))))))
```

3. A receiving-ticket is sent.

```
(@X (output (receiving-ticket.30
              (purchase-order.1))
            (to (ubik.dept.shipping))))
```

4. A query of the batch will show that it contains receiving-ticket.30.

```
(@Q (ubik.dept.shipping
      (input (batch (receiving-ticket.?x
                      (purchase-order.?po))))))
```

The reply is:

```
(receiving-ticket.30
  (purchase-order.1))
```

5. A notification-of-receipt is sent.

```
(@X (output (notification-of-receipt.2
              (purchase-order.1))
            (to (ubik.dept.shipping))))
```

6. A query of the batch will show that the receiving-ticket is no longer at the shipping department.

```
(@Q (ubik.dept.shipping
      (input (batch (receiving-ticket.?x
                      (purchase-order.?po))))))
```

The reply is `nil`.

7. A query of the purchase-order file will show that it contains a purchase-order.

```
(@Q (file.payable.purchase-order
      (purchase-order.?X)))
```

The reply is:

```
(file.payable.purchase-order
  (purchase-order.1))
```

A.3.9 Tapeworms

A tapeworm can be attached to a configurator. A when-modified tapeworm will trigger whenever the configurator to which the tapeworm is attached or any configurator below it is modified as shown in figure A.10.

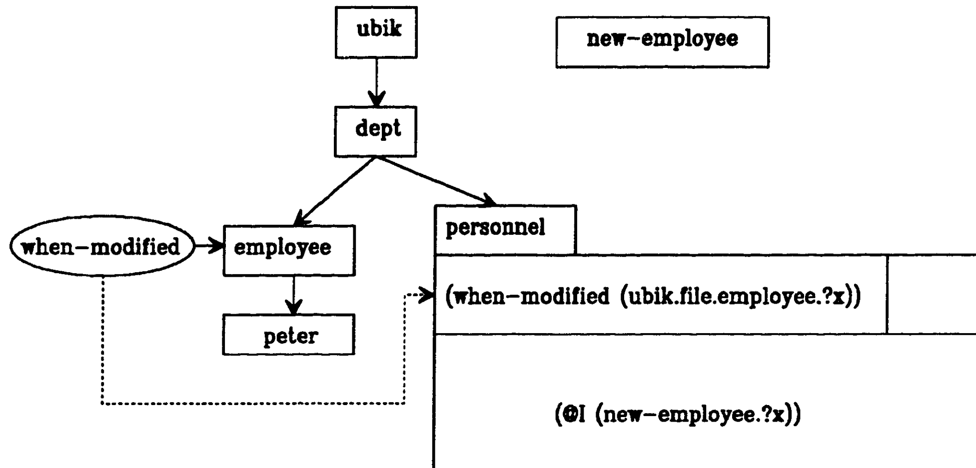


Figure A.10: When-modified tapeworm is installed in on a configurator. Installation is a two step process. The tapeworm is installed on the configurator as indicated by its name (ubik.dept.personnel). The When expression in its input section specifies the configurator which it is monitoring (ubik.file.employee). The tapeworm reference is installed in the monitored configurator. When triggered, a message is sent to the tapeworm.

The Ubik code for this example follows:

1. A organizational network is built.

```
(@I (ubik.file.employee.peter))
```

2. A tapeworm is added which will trigger when the network is modified. The input section contains the configurator which is being monitored. The tapeworm itself is placed on dept.personnel. When the tapeworm is triggered, it will build a configurator called new-employee, which will contain the employee configurator which caused the triggering action.

```
(@I (ubik.dept.personnel
      (input (when-modified (ubik.file.employee.?x)))
      (body (@I (new-employee.?x)))))
```

3. The employee configurator is modified with the addition of employee.jill.

```
(@I (ubik.file.employee.jill))
```

4. The tapeworm is triggered and is added to the new-employee configurator as this query will show.

```
(@Q (new-employee.?x)))
```

The reply is:

```
(new-employee.jill)
```

Tapeworms can be combined with batching. In the following example the tapeworm will trigger when the employee file is modified and a record-employee message is received for the employee.

1. The tapeworm is created.

```
(@I (ubik.dept.personnel
      (input (and (when-modified (ubik.file.employee.?x))
                  (record-employee.?x)))
      (body (@I (new-employee.?x))))))
```

2. The employee file is modified. The modification will be batched at the tapeworm.

```
(@I (ubik.file.employee.peter)))
```

3. A record-employee message is received for the employee.peter. This will cause the tapeworm to trigger.

```
(@X (output (record-employee.peter)
            (to (ubik.dept.personnel)))))
```

4. A query of the new-employee configurator will now show that new-employee.peter is there.

```
(@Q (new-employee.?x)))
```

The reply is:

```
(new-employee.peter)
```

A when-deleted tapeworm will be triggered when a configurator is deleted. In this example, when the deletion occurs, the tapeworm will create a changed-employee configurator.

```
(@I (ubik.dept.personnel
      (input (when-deleted (ubik.file.employee.?x)))
      (body (@I (changed-employee.?x))))))
```

A when-updated tapeworm will be triggered when a configurator is updated. In this example the changed-employee configurator will be created when an employee file is updated.

```
(@I (ubik.dept.personnel
     (input (when-updated (ubik.file.employee.?x)))
     (body (@I (changed-employee.?x))))))
```

A when-referenced tapeworm will be triggered when a configurator is referenced. In this example the referenced-employee configurator will be created when an employee file is referenced.

```
(@I (ubik.dept.personnel
     (input (when-referenced (ubik.file.employee.?x)))
     (body (@I (referenced-employee.?x))))))
```

A.4 Implementation

The initial framework for the implementation was based on the Logic Programming system of Abelson and Sussman [8]. This system uses Scheme, a sequential language, for its implementation. The system unifies queries, which are patterns containing variables, against a database of relations. The unification results in frames of variable bindings. These frames are gathered together in streams. Complex queries are decomposed into multiple simple queries. Each simple query is sequentially evaluated, with the stream of frames providing communications between the queries. A database of rules also exists. The query specified by the user must either match a relation or a rule within the database, to be successfully processed. Rules are treated as subgoals of the original query. The rules eventually decompose into simple queries against the database.

Ubik's framework differs from Abelson's and Sussman's as a result of the more complex Ubik language and the parallel implementation. The language processes distributed databases consisting of configurators rather than relations. Rules are replaced by messages to configurators and tapeworms attached to the configurators. A complex Ubik action is decomposed into simpler actions, all of which can execute in parallel. The parallelism results in multiple streams of frames rather than the single stream in the Abelson and Sussman system. Ubik uses the fine grained parallelism supported by the underlying Actor system [51] to increase the parallel processing. Parallelism comes from the following sources:

- The message traffic between configurators, as specified in the application. Each configurator which receives a message can execute in parallel.
- The tapeworms which are triggered as side-effects of referencing the configurator.
- The parallel processing of the *and expressions* and *or expressions*.

- The Ubik interpreter, which is written in the parallel language Acore. The Acore written functions, such as configurator searching and unification, execute in parallel.

The Acore language, which is used to implement Ubik, is a parallel object-oriented language. The Ubik implementation consists of a collection of Acore objects and functions which reference these objects. The following are the sections which describe the implementation.

- Section A.4.1 Basic-objects - the objects which comprise Ubik.
- Section A.4.2 Front-end - the functions which communicate with the user and transform the user input into Ubik objects.
- Section A.4.3 Evaluator - the functions which execute a configurator.
- Section A.4.4 Model and networks - The functions which build models and networks.
- Section A.4.6 Unification - the unification functions.
- Section A.4.5 Tapeworms - the tapeworm functions.
- Section A.4.7 Utilities - the utility functions.

A.4.1 Basic Objects

The Acore objects which implement the basic Ubik objects are divided into memory, configurator, and communicator objects. An object consists of acquaintances and methods.

Memory

Ubik networks are built in models which reside in Ubik memory. The memory object has the following acquaintances:

1. **Name** - the name of the memory.
2. **Lock-controller-actor** - manages the locks on networks. It supports read and write locks. It accepts messages with continuations. The messages specify whether a read or write lock is requested. The lock requested is set if possible; if not, the request is queued. When a lock is set, the continuation within the lock message is executed. The locks are managed as follows·
   (a) read lock message - if there is no write lock set or queued, then the read lock is dispatched. If there is a write lock set or queued, the read lock message is queued.

(b) write lock message - if there is a read lock or write lock set, then the write lock message is queued. No more read locks will be initiated once a write lock is queued.

(c) read lock removed - if there are no more read locks and a write lock is queued, it is dispatched.

(d) write lock removed - if there are write locks queued, then one is dispatched, if not, and read locks are queued, then all the read queued read locks are dispatched.

3. **Tapeworm-scheduler-actor** - manages the scheduling of triggered tapeworms.

4. **Memory-contents** - location of the network.

The memory object has the following methods:

1. **Index** - methods to maintain a two-level index.

2. **Instantiate** - methods to instantiate the network of configurators within the memory.

The memory is divided into models. Each model has a top-node to which all the configurators within the model are attached, and a current-context which is the default position within a network.

Configurator

The configurator is the central Early Ubik object. Networks are composed of collections of attached configurators. A configurator can be attached to another configurator at the lattice-in, lattice-out, and body acquaintances. A configurator attached to the lattice-in acquaintance is a parent configurator. Configurators attached to the lattice-out acquaintance are children configurators. Configurators attached to the body acquaintance are nested configurators.

The configurator object has the following acquaintances:

1. **Name** - Name of configurator.

2. **Input** - Input section object.

3. **Output** - Output section object.

4. **Body** - Body, contains the linked configurators or nested actions.

5. **Lattice-in** - Parent configurator for the linked network of configurators.

6. **Lattice-out** - Children configurators in network of linked configurators.

7. **Nest-in** - Configurator whose body this configurator is in.

8. **Tapeworms** - Tapeworms which are monitoring this configurator.

9. **Status** - A tapeworm can be attached to a configurator which doesn't currently exist. The status of **inserted** is given to a configurator which exists. The status of **virtual** is given to a configurator which doesn't exist. A virtual configurator is either a deleted configurator which has not yet been expunged, or a tapeworm monitored configurator which has not yet been installed.

The configurator object has methods for initiating the building, evaluating, and instantiating of itself. These functions will be described later.

Input Section

The input section object is referenced by the configurator input acquaintance. It has the following acquaintances:

1. **Message** - Pattern to match the incoming message.

2. **Batch** - The batch associated with the input. The batch contains an entry for the input section message, or multiple entries when the input section message is an *and expression* or an *or expression*. In the latter case, an entry is kept for each expression within the *and expression* or *or expression*. Each entry in the batch will contain the environments which result from unifing the input section message with the incoming message. See section A.4.3 for a detailed explanation of the use of the batch.

Output Section

The output section object is referenced by the configurator output acquaintance. It has the following acquaintances:

1. **Message** - Output message pattern.

2. **To** - Output message destination.

3. **Reply** - Reply message pattern.

Communicator

A communicator object transfers messages between configurators. It contains the following acquaintances:

1. **Operation** - specifies whether the communicator message is for a tapeworm or non-tapeworm.

2. **Message** - message being sent.
3. **To** - destination configurator.
4. **Reply** - reply configurator.
5. **Send-environment-id** - unique id used to qualify the variables in the message being sent.
6. **Receive-environment-id** - unique id used to qualify the variables in the input section message of the destination configurator.
7. **Environment** - stream of frames which contains the bindings of the variables in the message. As a message is sent from one configurator to another, the environment grows. When a message starts returning to the customers of the configurators, the environment shrinks. This process is similar to the growing and shrinking of a subroutine invocation stack.

A.4.2 Front-end

These are the routines which support the interface to the end-user. The prototype used a very simple interface, in which the end-user embeds a linear Ubik expression as a parameter of a function named **ubik**. A slightly more complex interface routine would establish a Ubik listener, in which the end-user enters the linear Ubik expression. More complex interfaces would use 2-D pictures. All these interfaces would eventually require the following two routines:

Driver

ubik - function which accepts input from the end-user. The input is parsed, evaluated, and then instantiated.

Parser

parser - converts the linear language input into networks of configurators.

A.4.3 Evaluator

The evaluator executes Early Ubik actions. The basic evaluation cycle is as follows:

1. **input section evaluation** - An incoming message is unified with the input section message. The unification occurs with the environment transmitted with the incoming message. An augmented environment is created as a result of the unification.

2. **body section evaluation** - The body section either contains configurators or actions. If it contains configurators, the input section message is unified with the configurators in the body, using the environment from the input section. If the body section contains actions, they are executed by the appropriate action function.

3. **output section evaluation** - The output section contains the message that will be either transmitted to another configurator, or be returned to the configurator which sent the message causing this configurator to execute. The sender of the message to this configurator is called the customer of this configurator. A communicator is constructed to send the message. The environment created or augmented by the evaluation of this configurator is sent with the communicator.

Evaluate

Most of the evaluation routines have the following three parameters:

1. **configurator** - The configurator currently being evaluated.

2. **communicator** - The communicator which caused this configurator to be triggered.

3. **environment** - The environment in which the evaluation is taking place. An environment is a collection of variable bindings, as described in section A.4.6.

The routines which comprise the evaluator are as follows:

- **eval-ubik** - Evaluation of end-user entered configurators. The end-user either specifies the operation which is used to evaluate the configurator or a query operation is assumed. When the end-user entered configurator is parsed, the operation attached to the configurator is interpreted as the name of a configurator whose body is the end-user entered configurator. For example, `(@I (ubik.accounting))` is parsed as a configurator with name `@I` and body `ubik.accounting`.

  Eval-ubik invokes the eval-command function to evaluate the command, as specified in the configurator name. If the name of the configurator is not a command, then the eval-query function is invoked.

- **eval-non-installed-configurator** - When an evaluation operation @X is encountered on an action, this routine is invoked. If the configurator has an output section, eval-output is invoked or else eval-body is invoked.

- **eval-command** - Routine which examines the name of a configurator to determine if it is an operation. If it is, the appropriate operation routine is invoked. The operations, along with the function for processing the operation, are as follows:

  - **@I** - insert-command
  - **@D** - delete-command
  - **@U** - update-command
  - **@reset-memory** - @reset-memory
  - **@create-model** - @create-model
  - **@set-current-model** - @set-current-model
  - **@set-current-context** - @set-current-context
  - **@expunge-model** - expunge-model
  - **@X** - eval-non-installed-configurator
  - **@Q** - eval-query
  - **eval-lisp** - eval-lisp-expression
  - **not** - not-expression
  - **and** - and-expression
  - **or** - or-expression

- **eval-configurator** - Used by the evaluator routines to initiate the basic evaluation cycle. It invokes, in the following order, eval-input, eval-body, eval-output.
- **eval-body** - The body contains a collection of configurators. For each configurator the following occurs:
  - Eval-command is invoked. If the configurator's name is an operation, the function will invoke the appropriate operation function.
  - Unify-environment is invoked if the configurator name is not an operation. The input section message is unified to the configurator.

  All the environments produced by evaluating the body are combined by the or-environment function.

Input

The input section routines unify the incoming message with the input section message. The input cycle is as follows:

1. Unify the incoming message to the input section messages which are on the batch, using the environment associated with the incoming message. The batch contains multiple input section messages if the input contains an *and expression* or an *or expression.* Place the environment resulting from the unification in the batch onto the batch entry associated with the message. Also place a unique-id into all the new frames resulting from the unification.

2. Invoke and-environment or or-environment to all the environments in the batch. The resulting environment will represent the collection of messages which trigger the configurator. Each of the frames in the resulting environment contains a unique-id. Remove the frames on the batch with these unique-ids, since these frames are currently being processed, and thus no longer need to be batched.

An example of the input cycle follows:

1. A configurator containing an *and-expression* is placed into the network of linked configurators.

```
(@I (batch-example
      (input (and (a (po.?x))
                  (b (po.?x))))))
```

2. After parsing, the batch contains the following entries.

```
entry 1 - (a (po.?x))
environment 1 -

entry 2 - (b (po.?x))
environment 2 -
```

3. Message `(a (po.1))` arrives.

```
entry 1 - (a (po.?x))
environment 1 - ((*100 (?x 1)))

entry 2 - (b (po.?x))
environment 2 -
```

4. Message `(a (po.2))` arrives.

```
entry 1 - (a (po.?x))
environment 1 - ((*100 (?x 1)) (*101 (?x 2)))

entry 2 - (b (po.?x))
environment 2 -
```

5. Message `(b (po.3))` arrives.

```
entry 1 - (a (po.?x))
environment 1 - ((*100 (?x 1)) (*101 (?x 2)))

entry 2 - (b (po.?x))
environment 2 - ((*102 (?x 3))
```

6. Message `(b (po.1))` arrives.

```
entry 1 - (a (po.?x))
environment 1 - ((*100 (?x 1)) (*101 (?x 2)))

entry 2 - (b (po.?x))
environment 2 - ((*102 (?x 3)) (*103 (?x 1)))
```

7. And-environment (see section A.4.6) is invoked after each message is processed. With the current batch, it will produce the following environment.

```
environment 1 - ((*100 (?x 1)) (*101 (?x 2)))
environment 2 - ((*102 (?x 3)) (*103 (?x 1)))

and-environment - ((*100 *103 (?x 1)))
```

8. The entries from the batch with the unique-ids are removed.

```
entry 1 - (a (po.?x))
environment 1 - ((*101 (?x 2)))

entry 2 - (b (po.?x))
environment 2 - ((*102 (?x 3)))
```

9. The input section evaluation returns the environment `((?x 1))`.

The input section functions are as follows:

- **eval-input** - Invokes input-batch.
- **input-batch** - Invokes input-unify for the incoming message to each message on the batch. Places the resulting environments on the batch. Invokes input-execute and then input-return.
- **input-execute** - evaluates the input section's message. The pattern is in the format of a configurator. The action performed depends on the name of the configurator.
  - **and** - invoke input-and-execute
  - **or** - invoke input-or-execute
  - **not** - invoke eval-ubik
  - **eval-lisp** - invoke eval-ubik
  - else return the environment from the batch
- **input-return** - removes all the frames which are successfully unified from the batch.
- **input-unify** - invokes unify-environment to unify the incoming message to the input section messages.

- **input-unify-tapeworm** - When the incoming message is from a tapeworm, this function is invoked rather than input-unify. A message from a tapeworm has a different format than a non-tapeworm message.
- **input-and-execute** - and-environment is invoked to find the frames which are ready to execute.
- **input-or-execute** - or-environment is invoked to combine the frames which have been created by input-unify.

Output

Output routines send and receive messages. A message can be sent in the following ways:

1. Output operation - the output operation supplies the message and destination.
2. Output section - the output section message is used. If the output section supplies a destination, it is used. If not, the message is returned to the customer.
3. Input section, no output section - the input section message is used. There is no output destination; the message is returned to the customer.
4. No input or output section - the configurator is entered by the end-user. The driver routine will instantiate the output and display it on the end-user's display device.

An output message is packaged for sending in a communicator. The communicator contains the message, the current environment, the destination, and a send-id, and receive-id. The send and receive ids are unique ids needed to distinguish variables during unification. Each time a message is received by a configurator it is unified with patterns within the configurator. All the variables involved in the unification are qualified using the receive-id. When a message is sent from the configurator to another configurator, a new receive-id is created. The receive-id for the current configurator becomes the send-id in the communicator. When the message arrives at a configurator and is unified with the input section message of the configurator, the incoming message is qualified with the send-id and the input section message is qualified with the receive-id.

The `to` attribute specifies the destination of the message. It specifies a network path. For example `to a.b.c` would send the message to configurator `c` on path `a.b`. There can be multiple `to` attributes which will send the message to multiple destinations. A `to` attribute can contain variables. For example `to a.b.?` would send the message to all the child configurators of `b`. `(to a.b.?x)` would send the message to the configurators whose

names are bound to ?x in the environment. Before a message is sent, the **to** attribute is instantiated to resolve all its variables.

The message is not sent to the output section or statement directly to the configurator; it is sent to a Ubik memory object. Each memory object has a receive function named ubik-receive. The receive function determines the model, network, and configurator which will receive the message. It invokes eval-configurator to start the evaluation process.

If there is no **to** attribute, then the output message is returned to the customer. The customer is actually implicit within the underlying Actor system. A version of Ubik implemented on a non-actor system would have to maintain the customer within the communicator. Ubik, as opposed to Early Ubik, explicitly maintains the customer.

A reply attribute in the output section, or on an output operation, is processed when the output is returned to the customer. The reply message is unified to the output message. During unification, the reply is qualified with the sent-id and the output message is qualified with the receive-id.

The output routines are as follows:

- **ubik-send** - sends a message to each destination.
- **ubik-receive** - receives a message. Determines the configurator within a network to which the message was sent, and invokes eval-configurator for that configurator.
- **eval-output** - if there is an output section, then its message is used; if not, the message in the input section is used. If there is a destination, eval-output-transmit is invoked, or else eval-output-return-environment is invoked.
- **eval-output-return-environment** - if there is a reply message, the reply message is unified to the returned output message. The environment is then returned to the continuation of the sending configurator. The continuation is maintained by the underlying Actor system.
- **eval-output-transmit** - the communicator is set up for sending to a destination. Ubik-send is then invoked.
- **instantiate-to** - instantiates the **to** message. All variables must be instantiated in order to determine the message destinations.
- **instantiate-lattice** - finds the configurator in the network to which the message is sent. Messages can be sent to destinations specified by wildcards. This function finds all the destinations implied by the wildcard.

Expression

Expressions can appear within an input section or body section. The processing of the *and expression* and *or expression* within the input section has been previously described. These are the functions for processing the expressions within the body section:

- **and-expression** - invokes eval-ubik in parallel for all the conjuncts of the and-expression, except for `eval-lisp` and `not` conjuncts. They are processed sequentially after the other conjuncts have returned. Eval-ubik will return an environment for each conjunct. The environments are combined by the and-environments function. This function takes the cartesian product of the conjunct environments. After this cartesian product takes place, a frame might have multiple copies of the same variable. If the values for a variable are incompatible, the frame is removed. An example follows:

```
(and (@X (output (a (b.?x))))
     (@X (output (c (d.?x)))))

conjunct 1 (@X (output (a (b.?x))))
conjunct 2 (@X (output (c (d.?x))))

conjunct 1 environment (((?x 1)) ((?x 2)))
conjunct 2 environment (((?x 3)) ((?x 1)))

cartesian product
(((?x 1) (?x 3)) ((?x 1) (?x 1))
 ((?x 2) (?x 3)) ((?x 2) (?x 1)))

results after incompatible values are removed
(((?x 1)))

results when instantiated
(and (a (b.1)) (b (d.1)))
```

- **or-expression** - invokes eval-ubik in parallel for all the disjuncts of the or-expression. The frames returned are combined by the or-environments function. An example follows:

```
(and (@X (output (a (b.?x))))
     (@X (output (c (d.?x)))))

disjunct 1 (@X (output (a (b.?x))))
disjunct 2 (@X (output (c (d.?x))))

disjunct 1 environment (((?x 1)) ((?x 2)))
disjunct 2 environment (((?x 3)) ((?x 1)))

results after or-environments
(((?x 1)) ((?x 2)) ((?x 3)))))

results when instantiated
(or (a (b.1)) (a (b.2)) (a (b.3))
```

- **not-expression** - invokes eval-ubik for the expression within the not. If it returns a failure, then the not-expression succeeds. If it returns a valid environment, then the not-expression returns a failure. An example follows:

```
(and (a (b.?x))
     (not (c (d.?x)))

expression 1 (a (b.?x)) results  (((?x 1)))
```

The not expression is processed after expression 1, using the bindings generated by expression 1.

```
expression 2 (a (b.?x)) with environment (((?x 1)))

results of evaluation (((?x 1)))

not expression 2 results - fail

and-expression results - fail
```

If the evaluation of expression 2 fails, then the example will succeed.

```
expression 2 (a (b.?x)) with environment (((?x 1)))
results of evaluation fail
not expression 2 results - nil
and-expression results - (((?x 1)))
results when instantiated
(and (a (b.1)) (not (c (d.1))))
```

- **eval-lisp-expression** - instantiates the expression within the eval-lisp expression. Lisp apply is then invoked on the instantiated expression. An example follows:

```
(and (a (b.?x))
     (eval-lisp (> 2 ?x)))

expression 1 (a (b.?x)) results  (((?x 1))(?x 2)))
```

The **eval-lisp** expression is processed after expression 1, using the bindings generated by expression 1.

```
instantiation of eval-lisp expression
(eval-lisp (> 2 1)))
(eval-lisp (> 2 2)))

and-expression results - (((?x 1)))

results when instantiated
(and (a (b.1))
     (eval-lisp (> 2 1)))
```

Query

Query supports direct reasoning over the structure of a network. A configurator pattern specified in the query is unified with the configurators in a network which match the pattern. For example, the following queries can be issued: find all the configurators which have an input section which accepts orders; find all the configurators which send an order; find the contents of the batch in a configurator. Query is implemented by the following routines:

- **eval-query** - finds the configurator in a model which corresponds to the specified configurator. The following routines are then invoked in order: query-lattice, query-input, and query-output.
- **query-lattice** - unifies the body of the query configurator to the body of the configurator in the model.
- **query-input** - if the input section contains a batch expression, then invoke query-input-batch, or else invoke query-input-unify.
- **query-input-unify** - unifies the query input message with the model input message.
- **query-input-batch** - unifies the query input message pattern with the messages in the batch.
- **query-output** - unifies the query output message pattern with the model output message.

A.4.4 Model and Networks

A memory consists of multiple models. Each model has the following attributes:

- model-name - the name of the model.
- current-context - the path name of a location within a network. The following example illustrates a path name into a network.

```
path name - a.b.c

network - (a (.b.c.d) (.e.f.g))
```

- top-node - the root of the networks within a model.

Model

The routines in this section locate and walk networks within a model. The following examples illustrate the way configurators are found in models given a path.

```
(top-node (a (.b.c.d) (.e.f.g))
          (x (.b.c.d))

current-context (a.b.c)

path (x (.b.c)) - locate 'c' configurator on 'x' path

path (.c) - locate 'c' configurator on 'a' path using context

path (.d) - locate 'd' configurator on 'a' path using context
```

- **@create-model** - creates a model with the specified name in memory.
- **@set-current-model** - sets the default model in memory to the specified model.
- **@get-current-model** - retrieves the current model location.
- **@set-current-context** - sets the current context in the current model.
- **context-walk-back** - walks backward from the specified position within the context.
- **find-actual-top-node** - finds the top-node of a network.
- **@get-current-context** - gets the current context from the current model.
- **@reset-memory** - resets memory.

- **@top-node** - gets top node of current model.
- **get-max-level** - when searching for a configurator within a network, more than one configurator can have the same name. This routine returns the longest path to a configurator with the same name. The longest path represents a configurator with the most context. If multiple paths have the same length, then an error is returned. For example:

```
(top-node (a (.b.c.d) (.e.f.g))
          (x (.b.l.c.d))

path (.c) would return (x.b.l.c) because path is longest.

path (.b) would return an error because two paths are
          the same length.
```

- **find-position-in-model** - given a network, this finds the path to a specified configurator within the network. There are many utility functions used by this function, which walk and match networks. The parameters of this function are as follows:
  - **top-node** - top node of model, or position where search starts.
  - **configurator** - configurator being located.
  - **use-context** - if true, then use current context in model to locate position.
  - **status** - find position on one of the following: inserted and virtual nodes, or inserted nodes. Virtual nodes are nodes created to contain tapeworms for which a configurator does not yet exist.

There are a collection of network matching utilities used by find-position-in-model. The names of the functions are as follows:

```
lattice-first-match
lattice-equal
lattice-last-equal
lattice-bottom
lattice-backward-visited
node-equal?
find-lower-configurators
find-higher-configurators
find-deleteable-paths
copy-lattice-between-name-and-configurator
find-on-path
copy-lattice-between-configurators
```

Network

These are the routines which insert, delete, and update a configurator within a network. These routines have to locate the configurator, perform the operation, and trigger any *when tapeworms*.

A modification of a network requires multiple configurators to be modified. To assure the coherence of the change, a write lock is placed on the network. This write lock also reduces the chance of deadlock, which can occur if two modification are using the same configurators. This deadlock is a result of the low level locking which automatically occurs on serialized objects within the underlying actor system.

The network functions are as follows:

- **lattice-command** - main modification routine. Instantiates configurator to be processed. Places a write lock on the network. Invokes one of the network command functions. After the command function has completed, the write lock is released and any tapeworms which were triggered as a result of the modification are dispatched.

- **insert-command** - inserts a configurator into the current model. This function performs the following: invokes command-in-body to find the location and the perform insertion; command-in-body returns path to inserted configurator; marks all configurators along path inserted; invokes collect-tapeworms to find all the tapeworms on the insert path; invokes filter-tapeworms to keep only the when-inserted and when-deleted tapeworms, and schedules the tapeworms for activation.

  If the configurator to be inserted is a tapeworm, insert it. The input section of a tapeworm specifies the configurators which it can trigger. These configurators are found, and the tapeworm reference is placed in the configurators acquaintance. The configurator, which the tapeworm references, might not be currently installed. Configurators are created to hold the tapeworm references. These configurators are marked virtual, since they were not explicitly created. If they are explicitly inserted later, they will be marked inserted.

- **delete-command** - deletes a configurator from the current model. The last configurator in the network path specified will be deleted, along with all the configurators below it. Deletion consists of marking the deleted configurators with the status virtual. They will be removed by the @expunge-model function. Command-in-body performs the delete.

  If a tapeworm is deleted, its reference is removed from all the affected configurators. Tapeworms can be deleted implicitly, because they lie below the configurator being deleted.

  All triggered when-delete and when-modified tapeworms are scheduled.

- **expunge-model** - will remove all the configurators in network with status virtual, except the paths which lead to a tapeworm.
- **update-command** - updates last configurator in path. Command-in-body performs the update. Updating a tapeworm can change the configurators which will be triggered. All triggered when-update and when-modified tapeworms are scheduled.
- **command-in-body** - function which locates the place in a network to modify, and performs the modification. It returns all the configurators along the path to the modification. The following are the routines which it invokes:
  - **not-in-model** - modification path not in model. A new network is added to the model.
  - **not-in-context** - modification path is not in current context of model.
  - **not-completely-matched** - add to the middle of a network path.
  - **complete-match-no-body** - completely matches a current network path.
  - **complete-match-no-model-body** - completely matches a current network path; the input configurator has a body but the configurator in the network doesn't.
  - **complete-match-body** - completely matches the network; enter the body to start a new network search. The body of a configurator can be the anchor of multiple networks.
  - **delete-configurator** - delete a configurator.
  - **update-configurator** - update a configurator.
  - **reset-nest-in** - reset the nest acquaintance of a configurator to specify the configurator into whose body the configurator was inserted.

A.4.5 Tapeworm

A tapeworm is installed in a network. The input section message of the tapeworm is installed in the configurator which will trigger the tapeworm. When a triggering action occurs, the action which caused the triggering is sent as a message to the tapeworm. The eval-configurator function is used to execute the tapeworm.

A tapeworm is triggered as the result of a modification or a reference to a network. The tapeworm is discovered when the path to the modified or referenced configurator is being resolved. When a configurator is involved in the resolution, and has a tapeworm attached, that tapeworm is triggered.

The modification or reference network pattern becomes the message which is sent to the triggered tapeworm. For example:

```
model (top-node (a.b (.c.d) (.x.y)))

(tapeworm-1
  (input (when-modified (a.b.c))))

modified configurator - c

message to tapeworm-1 - (a.b.c)

modified configurator - b

tapeworm-1 not triggered.

(tapeworm-2
  (input (when-modified (a.b.?x)))

modified configurator - d

message to tapeworm-2 - (a.b.c.d)
```

The executing of a tapeworm will not cause the triggering of another tapeworm. This is a restriction of the prototype to prevent a tapeworm from indirectly causing itself to be reinvoked. This reinvocation can cause a tapeworm loop. In future versions of Ubik this restriction should be relaxed.

The tapeworm activation is delayed. This delay is necessary in the prototype implementation to reduce the occurrence of deadlock. The underlying actor system supports a low level locking system, in which each configurator can be locked itself when it receives a message. A modification of a network requires the modification of multiple configurators. If the tapeworm references these configurators and locks other configurators, the possibility of deadlock is high. To reduce deadlock, Ubik has a higher level locking scheme, in which a network is locked when a modification operation takes place. Any new modification operation is prevented from executing until the current one is completed. Tapeworms resulting from a modification are scheduled until the modification is complete. In future versions of Ubik, which contain adjustable locking granularity, this restriction should be eliminated.

The following are the collection of routines which schedule and dispatch tapeworms:

- **collect-tapeworms** - given a list of configurators, collects all the tapeworms on the list. The list is collected by one of the network functions while searching a network. For example, in the process of inserting a configurator into a network, all the configurators along the path from the top-node to the place where the insertion is to take place would appear on this list.

- **filter-tapeworms** - finds tapeworms in a collection of tapeworms which are monitoring the specified event. Removes the tapeworms which are for events which did not occur.
- **tapeworm-scheduler** - object which contains queue of tapeworms which have been triggered. Receives two messages: schedule tapeworm and dispatch tapeworm. A schedule tapeworm will queue the tapeworm; a dispatch-tapeworm will start the execution of all the tapeworms on the queue by invoking eval-configurator for each tapeworm.

A.4.6 Unification

The unification routines consist of the functions which perform unification, instantiation, and environment maintenance.

Unify

These are the routines which implement the unification, as described in section A.4.6. Unification consists of matching one pattern to another. The order is significant because all of the first pattern needs to match the second, but not all of the second need match the first. In this section the first pattern is called *pattern* and the second is called *lattice*.

- **unify-environment** - externally called unification routine. If any of the patterns to be unified are installed in a network, then read lock is placed on the network. The function unify-lattice is invoked. Its parameters are as follows:
  - pattern - pattern to be unified to lattice parameter.
  - e1 - id, which is used to qualify the variables in pattern.
  - ins1 - specifies whether the pattern parameter is installed in a network.
  - lattice - pattern which will be unified with pattern parameter.
  - e2 - id, which is used to qualify the variables in lattice.
  - ins2 - specifies whether the lattice parameter is installed in a network.
  - environment - stream of variable bindings.
  - operation - input message type currently being processed. The types are tapeworm and non-tapeworm. This is needed so that the unification which occurs when processing a tapeworm does not trigger other tapeworms.
- **unify-lattice** - The pattern configurator is compared to the lattice configurator. The matching is as follows: a constant match results in examining the children of the two configurators. A failure occurs

if two constants do not match. Each path in the pattern configurator must match a path of the lattice configurator. The function returns an environment with new bindings or the symbol `fail`. The unify-frame function is invoked for each frame in the environment.

- **unify-frame** - recursive routine which walks the lattice network for each path of the pattern network. The pattern configurator can be a list of configurators. For example, when walking the network

```
pattern (invoice.24 (body ((customer.?x)))

lattice (invoice.24 (body ((customer.a)(salesman.24))))
```

  the function process-lattice is invoked for the path invoice.24, and then for the path customer.?x. Before invoking the function, instance-configurator is invoked to instantiate any configurator variables in the pattern.

  After unification is complete, merge-frames is invoked so that the configurator body, as represented by the configurator variables, is reassembled as follows:

```
initial ((invoice.24 (customer.a))
         (invoice.24 (salesman.mike)))

reassembled (invoice.24 (customer.a)(salesman.mike))
```

- **process-lattice** - recursive routine which walks the each path in the lattice network. For example, when walking the network

```
pattern ((customer.?x))

lattice ((customer.a)(salesman.24))))
```

  unify-frame-inner-body is invoked as follows:

```
pattern (customer.?x)

lattice (customer.a)
```

  and

```
pattern (customer.?x)

lattice (salesman.24)
```

  Before invoking unify-frame-inner-body, instance-configurator is invoked to instantiate any configurator variables in the lattice configurator.

- **instantiate-configurator-variables** - substitutes the configurators which are values of the variables for the variables. This supports the indirect reference facility of the configurator variables, as shown in section A.3.4.

- **unify-frame-inner-body** - continues recursive walk down network. If pattern and lattice configurators are equal, then invokes either unify-frame or unify-body to continue the walk down the lattice as follows:

  ```
  pattern (invoice.?x (body (customer.a)))
  lattice (invoice.a  (body (customer.a)))
  ```

  Invoke unify-equal? to match invoice. If equal, move down the lattice and invoke unify-frame with

  ```
  pattern (?x (body (customer.a))
  lattice (a  (body (customer.a))
  ```

  When the body is the next move down the lattice, unify-body is invoked.

- **unify-body** - invokes unify-lattice to continue the network walk with the body of the pattern and lattice.

- **unify-equal?** - matches the name of the pattern configurator with the name of the lattice configurator. The possible names are as follows:

  1. ? - wildcard
  2. ?? -configurator wildcard
  3. ?x - variable
  4. ??x - configurator variable
  5. x - symbol
  6. 1 - number

  The wildcards and variables will match anything. When they match, unify-extend-frame is invoked to place the variable in the frame. If both the pattern configurator and lattice configurator contain a symbol or number, they must be equal to each other or a unification failure results.

- **unify-extend-frame** - will invoke extend-frame with the new variable binding to place the variable in frame. If the variable is already bound in the frame, then unify-equal? is invoked to check whether the bindings are compatible. Process-reference-tapeworms is invoked to check whether reference tapeworms should be triggered. Free-for is invoked to see if the variable name occurs in the configurator matched to the variable.

- **free-for?** - checks to see if a variable name occurs in the configurator matched to the variable. For example, the following configurator contains an invalid variable name occurrence:

```
pattern (?x)
lattice (a (b.?x))
```

- **process-reference-tapeworms** - if the pattern configurator or the lattice configurator are installed in a network, they are checked to see if any when-reference tapeworms are attached. If they are, tapeworm-when-referenced is invoked to schedule the tapeworm.

Environment

An environment has the following structure:

```
<environment> := (<frame>*)

<frame> := (<bind>*)

<bind> := (variable id (<value-list>*)

<value-list> := (value id configurator)
```

A variable is a configurator-variable **??x** or a variable **?x**. An id is a unique number which qualifies the variable. Value is the variable, symbol, or number which is the value of a variable. The value is the name of a configurator object. Configurator is the configurator object which contains the name.

- **bind-in-frame?** - determines if a variable with corresponding id is in a frame.
- **extend-frame** - adds a value-list to a frame.
- **extend-environment** - adds a frame to an environment.
- **lookup-in-frame** - finds the value list associated with a variable in a frame. If the value is a variable, keeps searching until a constant value, if any, is found.
- **instantiate-environment** - when a configurator returns a message to its customer, the variables in the environment for that configurator will no longer be used. This routine removes all these variables. Before these variables are removed, any constant values indirectly associated with these variables will replace them.
- **or-environments** - combines multiple environments into one environment.

- **and-environments** - Parallel *and expression* create multiple environments which must be combined. The combination consists of taking the cartesian product of frames in the environments. Frame-cartesian-product is invoked.

- **frame-cartesian-product** - takes a list of environments and returns the cartesian product of their frame-lists. For example:

  ```
  in ((frame1 frame2) (frame3 frame4))

  out ((frame1-frame3) (frame1-frame4)
       (frame2-frame3) (frame2-frame4))
  ```

  If the resulting frame has a non-configurator variable with more than one value, then the frame is deleted. A variable with more than one value would cause a unification failure if the *and expression* conjuncts were unified sequentially rather than in parallel.

- **merge-frames** - merges a list of frames into an environment. There are two types of merge. Single-frame is used by frame-cartesian-product to remove all frames with variables which have multiple values. Multiple-frame produces a new frame for each multiple value found for a variable.

In addition, there are multiple utility routines for searching and manipulating an environment. The names of these routines are as follows:

```
multiple-frame-handling
merge-configurator-variables
union-environment
merge-variables
merge-value-list
collect-variables-in-frame
collect-variables-in-bind
collect-frame-identifiers-for-env
collect-frame-identifiers-for-frame
remove-frame-from-environment
remove-all-frame-identifiers
reset-variable-env
single-frame-handling
empty-environment?
variable-wildcard?
configurator-var?
configurator-wildcard?
frame-identifier?
any-var?
```

Instantiate

These are the routines which replace the variables in a pattern with their values from the environment.

- **instantiate** - initiates instantiation of a configurator.
- **instantiate-output** - initiates instantiation of a top-level configurator which was created by the end-user using Ubik's linear syntax.
- **configurator-instantiate** - creates an instantiated configurator for each frame in the environment.
- **instantiate-name** - instantiates the name section of a configurator; invokes copy-and-instantiate to walk the network of which the configurator is a part.
- **copy-and-instantiate** - walks the configurator, instantiating the input, output, and body section. Invokes instantiate-name for the name of each configurator.

Lispify

Instantiates memory and models in varying levels of detail.

A.4.7 Utilities

Initialize

Initializes the Ubik system when it is loaded. Creates a memory object and an object for generating unique-id's.

Utility

Collection of functions which map over lists in parallel.

- **make-dot-name** - input is (a b c); output is a.b.c
- **add-dots** - subfunction of make-dot-name.
- **mapcar@** - parallel version of mapcar.
- **filter-mapcar@** - parallel version of mapcar which also removes specified symbol or number.
- **mapappend@** - parallel flatten.
- **filter-mapappend@** - parallel flatten which also removes specified symbol or number.
- **mapc@** - parallel mapc.

- **assoc@** - parallel assoc.
- **sort-configurators** - sorts a list of configurators by the configurator name.
- **merge-configurators** - merges and sorts two lists of configurators.
- **unique-id** - generates a unique-id. Used for generating environment ids.

A.5 Conclusion

Ubik is a high-level language. Structures such as models, networks, configurators, and tapeworms are built into the language. The rationale for this is that these constructs are fundamental in describing organizations and their applications. Unification over these structures is a powerful reasoning facility. It enables the organization to reason over its own structure, as represented in Ubik.

Prolog [21] and Concurrent Prolog [60] represent alternatives to Ubik. These languages use unification over relations, functions, and lists. The simplicity of these languages allows a small interpreter to be built to evaluate the languages. A hoped for advantage of Ubik over Prolog-based languages is that it is much easier to use in writing business applications. Ubik could have been written in Prolog or Concurrent-prolog; it is not clear that this would have made the implementation of Ubik easier than its current implementation in Acore.

Ubik has many facilities not present in Early Ubik, such as power, development, censors, distribution, and prototypes. A future prototype will need to implement these additional facilities. Early Ubik has configurator-variables which were discovered as a direct result of using the Early Ubik implementation to build applications. Ubik, as described in this thesis, does not have configurator variables. It will probably need them.

Efficiency of implementation was not an objective of the prototype. A future implementation where it is an objective might require further Ubik language development.

Appendix B

Related Work

This section contains descriptions of systems which raise the level of application development. The systems will be analyzed by describing how they fit into the Ubik framework. The Ubik framework is as follows:

1. **Structure** - What are the basic objects of the system? How are these objects combined? What is the level of support for composite objects? Does the system contain prototypes? Is there support for object versions?

2. **Distribution** - How are logical and physical distribution handled? How is context handled?

3. **Action** - What is the interactive end-user interface? Does the system support end-user programming, batch processing, parallel computation, bureaucratic paths, passive messages, and active messages?

4. **Tapeworms** - Does the system support monitors and censors? Is the tapeworm installation commutative and distributed? What are the operations which are supported?

5. **Questers** - Can a quester query a system's structure and content? Are distributed questers supported?

6. **Development** - Can the system's representation be reorganized? Is there support for message elimination, regrouping, reclustering, bureaucratic development, and prototype development?

7. **Power relationships** - What are the facilities for cooperative and competitive processing? How is focus of attention handled?

8. **Integration** - How do the various system functions interact with each other? Does the system support or require the use of multiple languages and subsystems?

The following systems are described using this typology: hypertext in section B.1, relational databases in section B.2, rule and frame based systems in section B.3, and semantic net systems in section B.4.

B.1 Hypertext

A hypertext system is used to provide a cross referencing capability between multiple text documents. This capability includes an end-user interface to display multiple documents, a linking facility to create the cross references, and a browser to display the links and move between text documents. Hypermedia is a generalization of hypertext in which pictures and sound, in addition to text, can be cross referenced. The first hypertext proposal, Memex, was described by Vannevar Bush in 1945, in the article *As We May Think* [16]. Bush hypothesizes that the human brain works by associations between its concepts. A hypertext system, to him, mimics the way the brain works. This view of human thinking, combined with the end-user interfacing philosophy known as *what you see is what you get* (WYSIWYG), has guided the way hypertext systems are constructed.

Superficially, hypertext systems do not seem to have much in common with application development and Ubik organizations. However, as hypertext systems become more complex and are integrated into the general computing environment, they are evolving into Ubik type systems. Halasz, in the article *Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems* [35], discusses this evolution as follows:

1. Search and query in a hypermedia network - search must include both the structure and content of the network.

2. Composites - augmenting the basic node and link model such that groups of nodes and links are treated as first class objects.

3. Virtual structures for dealing with changing information - the hypermedia model needs to be augmented with a notion of virtual or dynamically-determined structures.

4. Computation in (over) hypermedia networks - active computational engines for particular applications need to be integrated into the hypermedia model.

5. Versioning - a mechanism to deal with changes needs to be integrated into the network.

6. Support for collaborative work - the support of simultaneous multiuser access to a common network, and the social interactions involved in collaboratively using a shared network, must be improved.

7. Extensibility and tailorability - make it easy for the end-user to make small changes to the system with a minimal amount of effort.

Structure

The basic object in a hypertext system is a node, consisting of a piece of text, and links to other nodes. Figure B.1 is an illustration of the Notecard hypertext system [35]. This example contains three nodes which appear as windows on a display screen. The end user retrieves a node onto the display screen by selecting a boxed string of text. In this example, if the end-user selects the boxed text **US TNF Missiles** in the node **characteristics of TNF Missiles**, then the node **US TNF Missiles** appears on the screen. These two nodes are called file nodes. The FILE BOXES section of the file node provide a hierarchical index. The NOTE CARDS section of the file node provides a link to a non-file box node. If the boxed text **Capabilities of New Missiles** is selected in the NOTE CARDS section, the corresponding node will appear as a window on the screen. This node has a boxed text **Guidance of Pershing II** which links to another related node when selected.

A Browser is a window which gives a global view of the linked nodes in a hypertext system. Figure B.2 illustrates a browser for the file box nodes in the previous example.

Distribution

Hypertext databases are distributed as follows:

1. The database is distributed physically over multiple computers, but the end-user does not see the distribution. For example, the KMS [10] system uses the Sun NFS file system to link the distributed databases into one logical database, which the end-user manipulates. This is illustrated in figure B.3. In this example there are six computers communicating over a local area network. Four of these computers contain files which are part of the KMS hypertext database.

2. Multiple contexts are maintained in one hypertext database. For example, the Intermedia system [65] maintains multiple collections of links between text documents. Each collection is called a web. A database is viewed through a web, with only the links in the web currently being viewed on display. Intermedia only maintains one level of webs; there are no webs of webs. Figure B.4 illustrates a database with two webs. Web 1 displays links b and g in document A, and a and f in document B. Web 2 displays links f and g in document A, and g and f in document B.

3. Multiple versions can be maintained for one node in a database. Multiple versions of collections of nodes, as described in section 3.2 on software development, are not maintained.

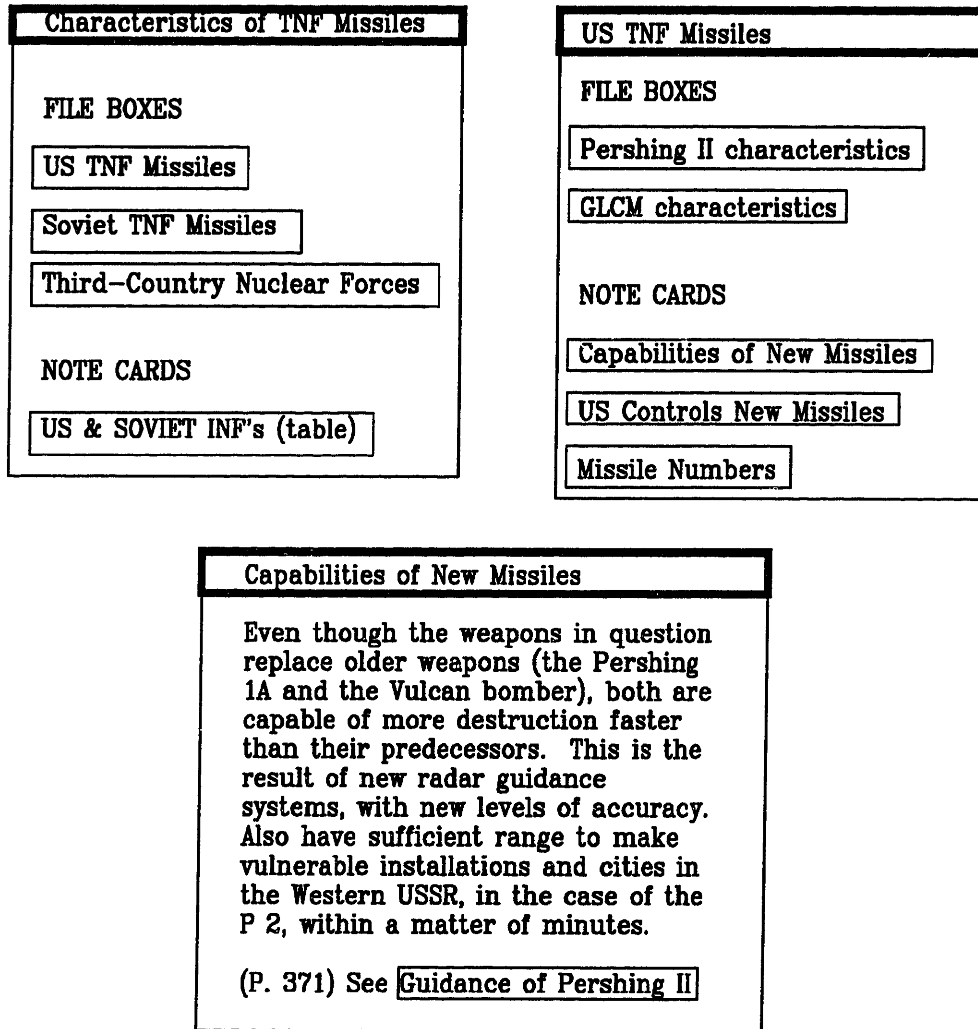


Figure B.1: Nodes in the Notecard hypertext system. The two upper nodes are file nodes, the lower one is a non-file node.

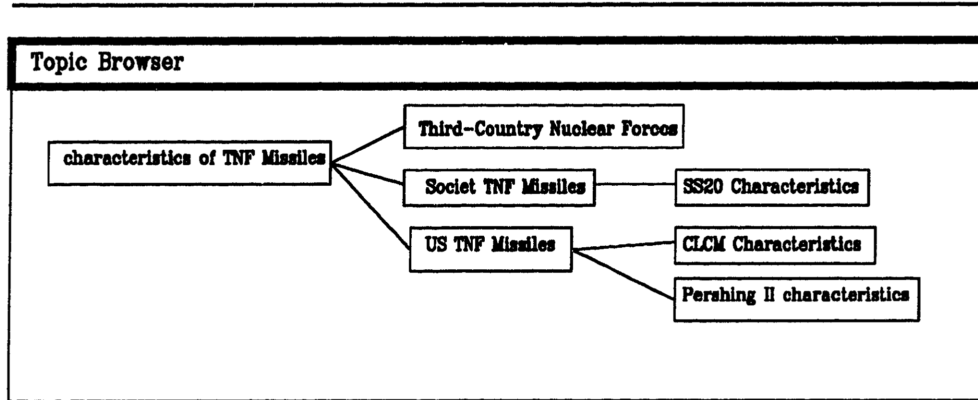


Figure B.2: Topic browser for notecard hypertext system.

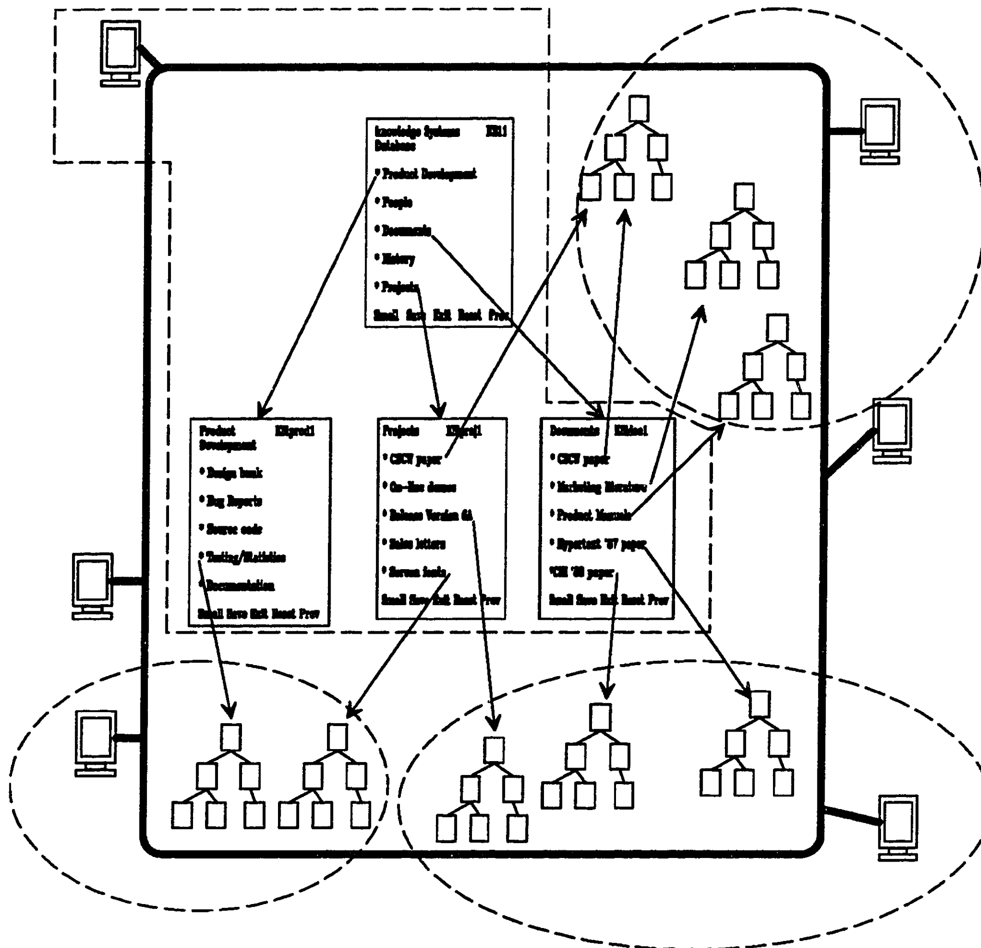


Figure B.3: The KMS hypertext system can be distributed over multiple computers linked by a local area network.

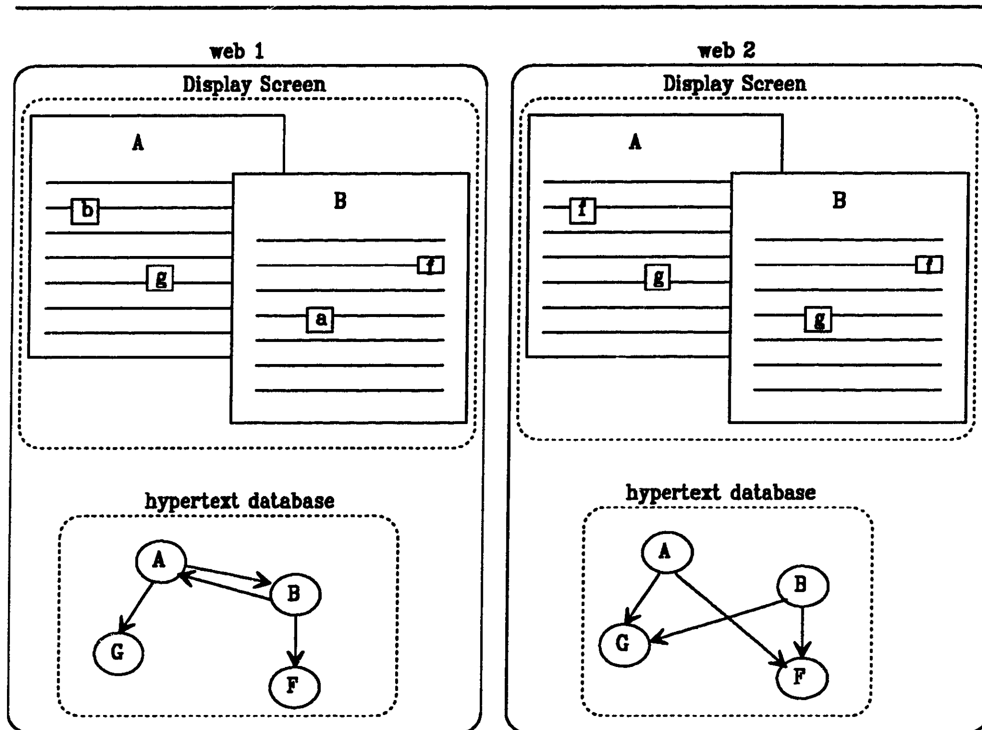


Figure B.4: Intermedia webs are used to maintain context in a hypermedia database.

Action

The nodes in a hypertext system are passive; they do not include programs. Programmable interfaces are provided in most hypertext systems. These interfaces are used by application subsystems to reference the hypertext database. Apple's Hypercard, which runs on the Macintosh computer, comes with its own programming language, which can make decisions based on the values filled into hypercard fields.

Tapeworms

Tapeworms are not supported in any of the systems.

Questers

Hypertext systems support many types of querying, some of which follow:

1. Browsers are an important part of hypertext systems. A browser provides facilities for the end-user to navigate through the network of nodes. Current research in hypertext systems involves creating browsers which help the user keep a frame of reference when navigating unfamiliar hypertext databases. Browsers are discussed further in the section on power relationships.

2. Graphical query languages would provide future hypertext systems with the ability to search a hypertext systems for specified linking patterns.

3. Link attributes, which exist on some current hypertext systems, allow a combination of semantic and structure queries. A query of this type could perform a structure query on only those links whose attributes are specified.

4. Text search treats a hypertext database as an unordered collection of text. The search for specified patterns can occur on the actual text, or on indexes built from the text.

5. Text and structure searches use a combination of text search and links. This type of search is discussed in the section on power relationships.

Development

Hypertext systems cannot be easily reorganized. The heart of the hypertext system is the explicit links between nodes. These links impose a structure on the system. There is not much information on the intent of the links, so any reorganization will result in lost hypertext information.

Power Relationships

Cooperative processing entails a group of people working on the same collection of nodes. A KMS node can be used as a mailbox, bulletin board, or discussion area. KMS users *grow* a conversation at a node in the database. The end-user views the conversation as text editing rather than message sending.

Focus of attention mechanisms in hypertext are as follows:

1. Browsers both provide a focus of attention on the nodes in which the end-user is interested, and a framework in which the end-user can view the complete database. A browser which does not focus attention can become unusable, as shown in figure B.5. In this example a global browser for part of an Intermedia database is shown.

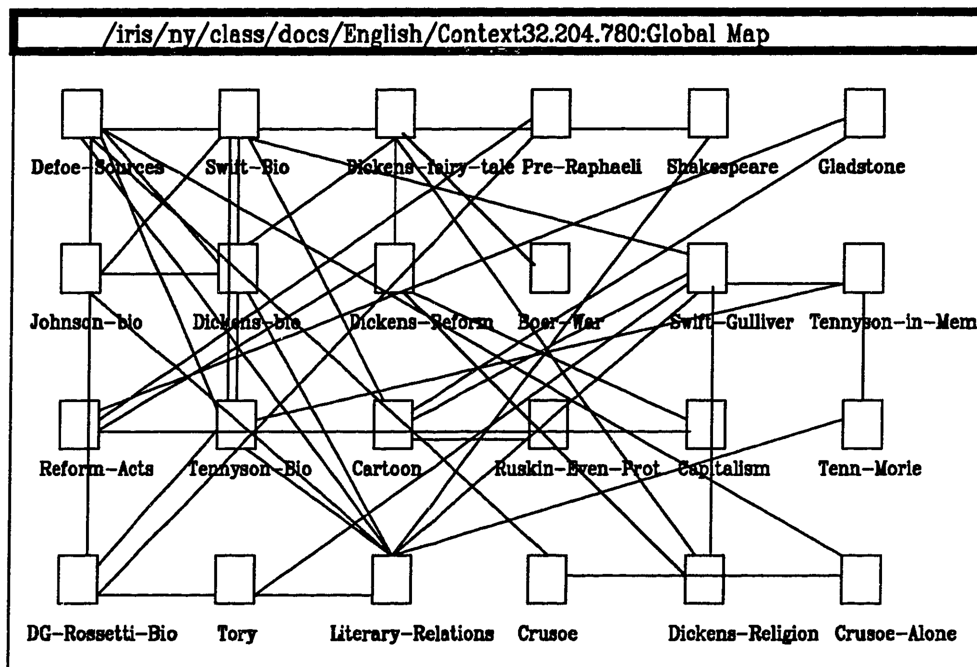


Figure B.5: An Intermedia database can become too complex for use with a non-focused global browser.

The focusing and framework mechanisms are as follows:

(a) Fish eye display focuses on a set of nodes and reduces the resolution and size of the surrounding nodes.

(b) Two-level windows provide a detailed window for the focused frames and a high-level window for the global view. The high-level window has an indication of the area the low-level window is displaying.

(c) The complexity of the database is reduced by displaying only the links for a specified web.

(d) The complexity of the database is reduced with subtree detail suppression, which replaces suborganizations of nodes with a common father node. A suborganization is expanded only when the end-user wants to navigate though it [30]. This is illustrated in figure B.6. In this example nodes are collected into sets. Links to nodes within a set are replaced by links to the set as a whole.

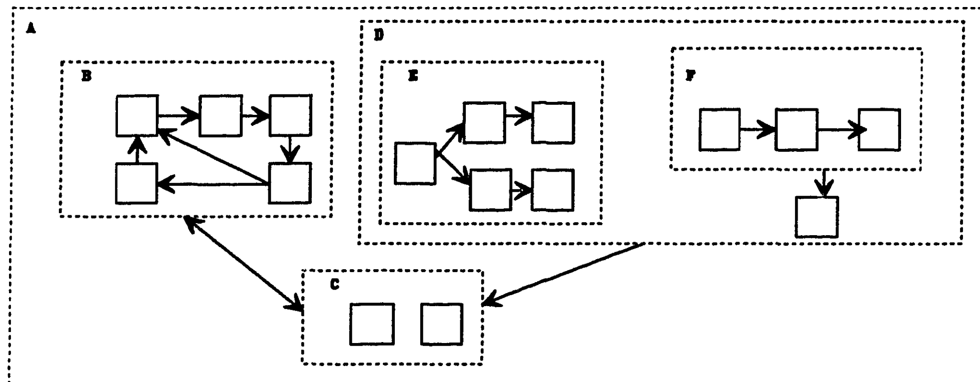


Figure B.6: A Browser suppressing links displayed by using subtree detail suppression.

2. Hypertext query processing should provide the end-user with *optimal* starting points for browsing. The query needs to use the node content, node context, and link semantics for finding these starting points. Out of research in searching for information in a hypertext medical handbook, Frisse developed the following principles for query processing [32]:

   (a) The utility of a card can be approximated by a computed numeric weight consisting of two components. The *intrinsic* component is the value computed from the number and identity of the query terms contained within the card. The *extrinsic* component is the value computed from the weight of immediate descendant cards.

   (b) The intrinsic card weight should be proportional to the number of times each query term occurs in the card, and inversely proportional to the number of cards that contain each query term.

   (c) The extrinsic card weight component should be inversely proportional to the number of immediate descendant cards. A card with many immediate descendant cards, but only one query term on one immediate descendant card, should have a lower weight than

docs a card with fewer immediate descendant cards and only one query term on one immediate descendant card.

(d) The optimal starting point for graphical browsing is the card with the highest weight. The next most *optimal* starting card is the card with the next highest weight that is not a descendant of any card with higher weights. If the next card is an immediate ancestor of any previously identified starting point card, the ancestor card should assume the descendant's role as a starting point card.

The results of Frisse's principles is illustrated in figure B.7. In this example a page from the medical handbook is shown. There are four links in this page. The labeled tree below the page shows the results of applying the principles. The path S1.IV.B.1 is the most promising path. The path S1.IV.E.1 and S1.IV.E.3 are the next most promising paths.

3. Hypertext can be used for problem exploration, as described in Conklin's Hypertext survey [22]. Synview [50] is a problem exploration system in which users post issues and arguments. The users of the system vote on the validity and relevance of a posting. The users can use the voting score to focus on the postings with the highest values.

Integration

Hypertext systems consist of the integration of an interactive user interface, links, and a browser which navigates the links. Other facilities needed in an application development system are missing or added on in an ad-hoc manner. The most important of the missing facilities is an integrated action system.

S1.1V.ENDOREACTRACHEAL INTUBATION

IV. Endotracheal Intubation and Tracheostomy. Endotracheal diameter of their lumen e.g. a no. 8 tube is one with an 8 mm lumen. Since the resistance to airflow is proportional to the fourth power of the tube radius, a large tube e.g.> no. 8 is preferable to minimize airway resistance and work of breathing. A large tube also permits easier suctioning and allows passage of the bronchoscope when bronchoscopy is indicated.

S 1.IV.A INDICATIONS. THE MOST COMMON

S 1.IV.B. ENDOTRACHEAL INTUBATION SHOULD BE

S 1.IV.C. TRACHEOSTOMY IS INDICATED WHEN

S 1.1V.E. PROBLEMS AND COMPLICATIONS

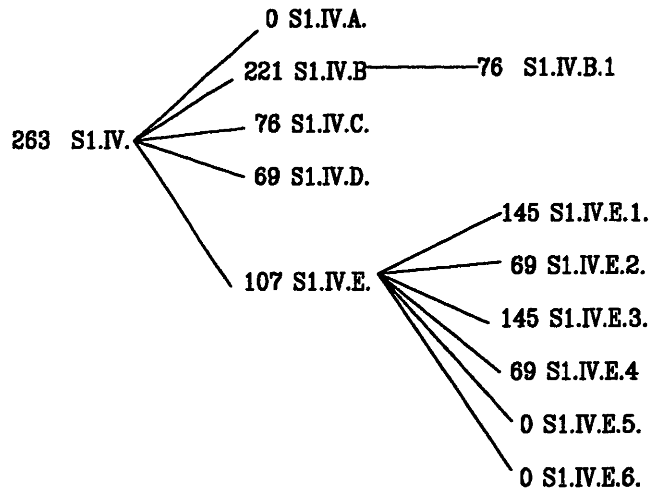


Figure B.7: Result of Frisse's principles in calculating promising browsing paths.

B.2 Relational Databases

The relational database model provides a tabular view of data along with operations for manipulating the tables. The operations have the power of first-order predicate calculus. This tabular view of data, provided by the model, has been used to develop end-user oriented application development systems. The System for Business Automation (SBA) [17,27,28,67] and its subset Query-by-Example (QBE) [66] are early examples systems of this type.

Paradox [7] is a product running on the IBM personal computer which is based on Query-by-Example. A Paradox query is shown in figure B.8. This query finds all the customers whose zip code is greater or equal to 90000 and whose credit is greater than 100000. The answers are selected from the customer table. A table in Paradox contains a name and columns. Operations written within the skeleton of the table define the query. In this example, the check marks indicate which columns are to be displayed in the answer. The selection expressions ¿=90000 and ¿100000 are placed in the Zip code and Credit columns respectively. Paradox returns the result in a table called ANSWER.

CUSTOMER	Cust ID	last Name	Init	Street	City	State	Zip	Country	Credit
		✓	✓		✓	✓	>=90000		✓ >100000

ANSWER	Last Name	Init	City	State	Credit
1	Connors	S	Belair	CA	900,000.00
2	Harris	J	Atherton	CA	750,000.00
3	Matthews	J	San Francisco	CA	1,250,000.00
4	McDougal	L	Seattle	WA	150,000.00
5	Montaigne	S	Bellevue	WA	450,000.00

Figure B.8: A Paradox query which finds all the customers whose zip code is greater or equal to 90000 and credit is greater than 100000.

Tables can be combined using a join operation. Join operations in Paradox are written as examples of how an end-user would manually look up the results in a table. Figure B.9 illustrates a join. The query in this example relates the customer id with the item he ordered after 6/1/88, and a description of the item. The customer id is in the orders table, and the description is in the products table. The tables are joined on the stock # column. This is indicated in this example by the use of the xyz symbol.

Structure

The table is the basic Paradox object. A table consists of rows and columns. Each row in a table must be unique. Each column must be of the same data type. The data types are alphanumeric, number, currency, date, and short

ORDERS	Order #	Cust ID	Stock #	Date	Emp #
		✓	✓ xyz	✓ >6/1/88	

PRODUCTS	Stock #	Description	Quant	Price
	xyz	✓		

ANSWER	Cust id	Stock #	Date	Description
1	2117	632	11/22/89	Portable suntan machine
2	2779	898	8/01/88	Matching panthers/leashes
3	3266	519	12/16/90	Robot-valet

Figure B.9: A Paradox query with a join. This query relates the customer id with the items ordered after 6/1/88 and a description of each item.

number. A family consists of a table and its related objects. These related objects are as follows:

1. Form - A table can be viewed through a form, as shown in figure B.10. In this example, the Luxury Gifts Department Customer Information Form displays a form for each row in the customer table. The user assigns the columns in the table to positions on a form by conducting an interactive dialog with the end-user.

   A form can be composed of two tables as shown if figure B.11. A two table form uses one table as a master table, and the other as a detail table. In this example the customer table is the master table and the bookord table is the detail table. The tables are linked through an interactive dialog on the cust id column.

2. Report - An object to display a table. A report can sort and group rows, calculate fields and totals, enter titles and headings, and arrange the table data in a variety of formats. Figure B.12 illustrates the specification of a report format. Figure B.13 shows the report generated by the format. A report format consists of bands. The bands are as follows:

   (a) Report band - indicates header for report.
   (b) Page band - indicates page heading and footings
   (c) Group band - indicates column that the report is being grouped on. The table being printed is sorted on the group column. When

CUSTOMER	Cust ID	Last Name	Init	Street	City	State	ZIP	Country	Credit
1	1386	Aberdeen	F	45 Utah Street	Washington	DC	20032		50,000.00
2	1388	Svenvald	I	Gouvernment House	Reykajavik			Iceland	1,250,000.00

```
LUXURY GIFTS DEPARTMENT
CUSTOMER INFORMATION FORM

******************************************
*                                        *
*  Customer Number:  1386                *
*                                        *
*                   Name: F. Aberdeen    *
*                                        *
******************************************

Address: 45 Utah Street
         Washington          DC 20032

Credit Limit:  $  50,000.00
```

Figure B.10: A Paradox form for viewing a row in a table.

CUSTOMER	Cust ID	Last Name	Init	Street	City	State	ZIP	Country	Credit
1	1386	Aberdeen	F	45 Utah Street	Washington	DC	20032		50,000.00
2	1388	Svenvald	I	Gouvernment House	Reykajavik			Iceland	1,250,000.00

BOOKORD	Cust	Date	Item #	Uol	Quant	Emp #
1	1386	8/21/87	1	M13	22	146
2	1386	18/20/89	1	M18	21	146
3	1386	10/20/89	2	M27	11	146
4	1784	5/05/88	1	I16	23	517

Viewing Bookord table with form F1: Record 1 of 7 (1–M Group)

LUXURY GIFTS BOOK CLUB
ORDER FORM

ID: 3128
Name: R. Elspeth, III
Address: 1 Hanover S.uare
London
England

Date	Item #	Vol	Quant	Sold by
2/18/90	1	I16	10	146
5/18/90	2	M64	11	146
5/18/90	3	E22	5	146
9/14/90	1	S09	14	775

Figure B.11: A Paradox form for viewing two tables. The customer table is the master table, and the bookord table is the detail table. The tables are linked on the cust id column, as specified with an interactive dialog with the end-user.

the value data in the group column changes, a total for that value can be printed on the report.

(d) Table band - the format of the table being printed. In this example the table has columns stock #, ID, Quant, Date, Emp #, and order #.

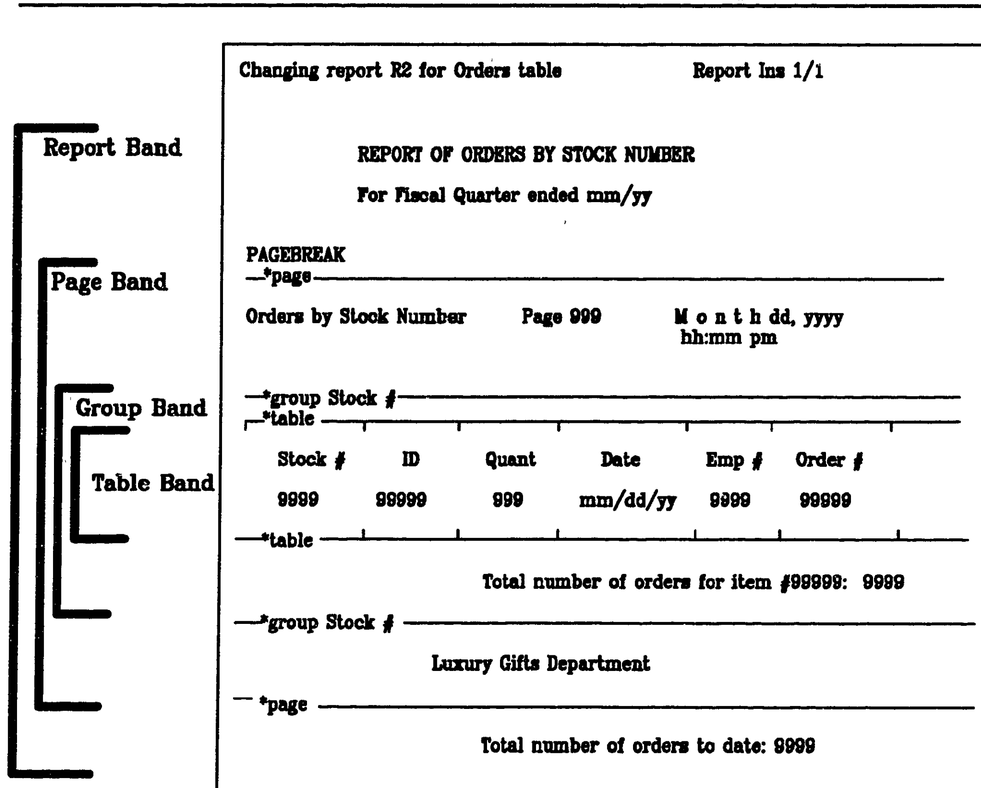


Figure B.12: Paradox format for printing a report. The report table is grouped on the stock # column. Group and page totals are printed.

3. Graph - A collection of objects which can display a table. Release 3.0 of Paradox supports the following types of graphs: stacked bar, standard bar, rotated bar, 3D bar, XY graph, area graph, line graph, pie chart, marker graph, and combined lines and markers. A bar graph is shown in figure B.14. In this example an answer table is generated, and then a crosstab table is generated from the answer table. A crosstab table takes the values of data in a specified column and makes them the names of the columns in the crosstab table. The graph labels are specified by an interactive dialog with the end-user.

4. Index - An object defined for each table column which facilitates quick retrieval of a value from that column.

REPORT OF ORDERS BY STOCK NUMBER

For Fiscal Quarter Ended 4/88

Orders by Stock Number   Page 1   April 28, 1988
10:00 am

Stock #	ID	Quant	Date	Emp #	Order #	
130	4277	1	2/28/87	775	2280	
		Total number of orders for item # 130:				1
235	7008	1	3/04/89	517	5119	
		Total number of orders for item # 235:				1
244	9004	5	9/04/89	422	3885	
244	5341	3	12/24/88	146	4492	
		Total number of orders for item # 244:				8
		Total number of orders to date:				10

Luxury Gifts Department

Figure B.13: The Paradox report for the report format shown in figure B.12.

MASTERRBK	Date	Employee	Customer	State	Home Group	Year	Total
		✓+	✓	✓	✓		✓

ANSWER	Employee	Customer	State	Home Group	Total
1	Christiansen	Aberdeen	DC	Manners	658.98
2	Christiansen	Aberdeen	DC	Manners	1,258.95
3	Christiansen	Aberdeen	DC	Manners	989.45
4	Morris	McDougal	WA	Investment	2,068.85

CROSSTAB	Employee	Estate	Travel	Manners
1	Chambers	1,139.05	1,707.15	8,543.20
2	Christiansen	3,714.75	2,068.25	4,555.80
3	Kling	2,307.60	479.60	1,528.75
4	Lee	5,934.60	1,558.65	0.00
6	Morris	2,067.45	3,774.50	1,168.75

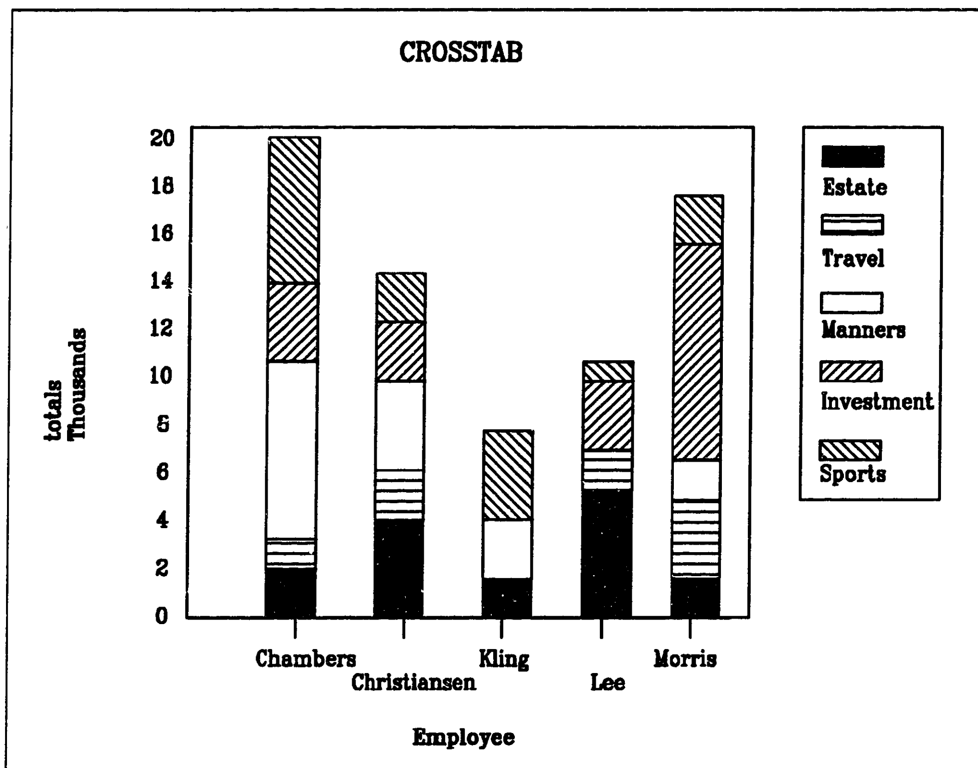


Figure B.14: A Paradox bar graph using a crosstab table.

5. Validity check file - A file which contains a constraint or check placed on values in a column.

6. Image setting file - A file which contains information on how the table is to be viewed.

A composite object is an object constructed from multiple tables. A form can be a composite object, consisting of two tables. A table itself can be a composite object, composed of multiple tables. Tables of these types can only be used for data entry. Data entered in this table are automatically placed into the tables linked to this table. A more general relational database concept, not supported by Paradox, is a *view*. A view is a table which does not actually exist; it is composed from underlying tables. The view can be queried as any other table and, to a limited extent, modified.

The use of composite objects are quite restricted in Paradox. Whereas tables can be used to create composite objects, composite objects cannot be used to create other composite objects. Tables can be queried using Query-by-Example. Composite objects can only be queried by using an interactive dialog or a function based language.

Distribution

Paradox can operate on files distributed over a local area PC network. The network supports the paths to the files. Paradox supports table protection and locking. Paradox does not support database transactions. A transaction ensures that a database operation, which consists of multiple table accesses or modifications, occurs as an atomic operation. Either the transaction is successful or it is aborted. An aborted transaction will remove all the database modifications made within the transaction.

Name context in Paradox is as follows: a table name is unique within a database, and a column name is unique within a table.

Action

Paradox supports both interactive and batch action. The interactive action consists of a menu interface which supports an end-user dialog. This interface can be used to create, browse, and query the Paradox objects. In browse mode the user can locate, display, and edit objects. A script can be made to package and repeat the user interaction. Query consists of forming Query-by-Example expressions which can be used to display and create objects. The interactive dialog can also be used to create custom interfaces to run particular applications. Batch processing consists of a C-type programming language called PAL. This language provides functions to reference the Paradox Objects.

Tapeworms

Paradox does not support general monitoring or censoring. It provides two types of censors: validity checks and referential integrity. Validity checks assure that the values in a column pass certain criteria. The validity checks are as follows: low value, high value, and one of a collection of specified values. Referential integrity is a guarantee of internal consistency for an object, even when the object is stored in two or more tables. General referential integrity is not supported. A limited type is supported in multitable forms.

Questers

Paradox supports an extensive set of query operations which are integrated with the Query-by-Example linking variables. The operations are categorized as follows: selection, projection, join, modification, calculation, grouping, set, and, or.

Development

Paradox supports extensive regrouping facilities. These facilities follow:

1. Query-by-Example queries support the creation of new tables out of existing tables. The new tables can be projections, joins, or aggregations of values in the existing tables.

2. Reports support sorting, column value breaks, and aggregating operations. With these operations, reports can be written with group headers and footers.

3. Graphs support crosstabing a table. A crosstab table is a table created from a relational table in which the data in one field is summarized by expressing it in terms of two other fields. Crosstabing involves selecting a first column whose values will supply the graph x axis labels, a second column whose values will supply the graph y axis labels, and a third column whose values will be the graph entries. The entries will either be the sum, min, max, or count of the values associated with the x and y graph axis.

Power Relationships

Cooperation in Paradox is through sharing of tables among multiple users. The support of sharing in Paradox is very primitive. Most modern relational database systems support transactions and fine-grain locking. Paradox does not support transactions, and its locks are at the table level.

Integration

Paradox's application development languages include a menu-based dialog language, a Query-by-Example query language, and a C-type programming language called PAL. The languages are well integrated. An end-user can start constructing his application using the dialog and Query-by-Example. He can drop down to the PAL programming language when he needs to express some application logic too complex for the higher level languages.

Query-by-Example is a powerful end-user programming language. Paradox treats Query-by-Example as just one of many interfaces, rather than as a unifying programming language for application development. In Paradox, Query-by-Example can only be used on tables, not other objects such as forms, reports, and graphs. An interactive dialog is used for querying these non-table objects. In Paradox, Query-by-Example is not used for directory management, specifying integrity constraints, or invoking actions. Ubik uses the Query-by-Example operations in a uniform manner for all the operations it supports.

B.3 Rule and Frame Based Systems

Model-based reasoning consists of representing the structure of an application area, defining the behavior of the application objects, and reasoning over the structure and objects. The most widely used model-based reasoning systems consist of rules and frames. The frames are used to represent the structure of the application, and the rules to reason over the frames. KEE [2,3] is the most complete system of this type. KEE consists of the following multiple interrelated subsystems:

1. Units - these are the KEE frames used to model an application. A unit consists of named slots which contain values.
2. Rule system - these are the objects which reason over the units. Rules monitor the slots within a unit. When the value of the monitored slot changes, the monitoring rule is triggered. A triggered rule can take action by invoking procedures and changing slot values.
3. Object-oriented programming - a way of taking action by sending a message to a slot. A program associated with the slot is executed when the slot receives the message.
4. Active Values - a way of taking action by changing the value in a slot. A program associated with the slot is executed when a value within the slot is changed.
5. Pictures - a subsystem which allows the building of interactive window displays.
6. Active images - a subsystem which provides graphic displays of the values in a unit.
7. Worlds - a context mechanism which maintains multiple versions of units.
8. Truth Maintenance - a subsystem which uses derivation rules and justifications to maintain consistent collections of units within a world.
9. TellAndAsk - a subsystem which provides English-like syntax for specifying rules.

Structure

The basic objects in KEE are rules and units. A unit contains slots. A slot contains facets, each of which contains values, as shown in figure B.17. The facets of a slot in KEE are as follows:

1. Value - value of the slot.

2. Comment - slot documentation.
3. Value class - Allowable values in a slot.
4. Cardinality max and min - number of values that are allowed in a slot.
5. Inheritance - how the slots within a unit are inherited from superclass units.
6. Method - method which is invoked by the receipt of a message.
7. Active value - method which is invoked when the value of the slot changes. This is further described in the section on tapeworms.

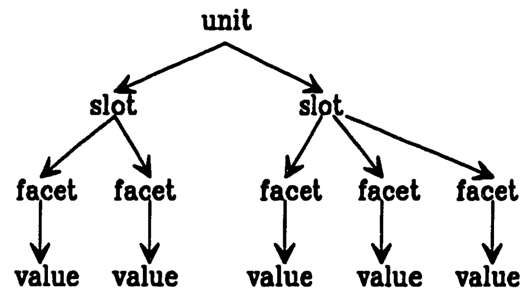


Figure B.15: A unit is composed of slots. Each slot is composed of facets with values.

Figure B.16 is an example of an automobile unit. The unit contains three slots: color, number.of.doors, and owner. The slots have a value of unknown.

Units can be organized into a class structure. A subclass inherits the slots and facets from superclass. In figure B.17, sedans and station.wagons are subclasses of automobiles. They inherit the three slots from the automobiles class. The sedans unit gives the number.of.doors slot a value of 4. The station.wagons unit gives the number.of.doors slot a value of 5 and adds the length.of.cargo slot. The unit marys.car is a subclass of sedans. This unit inherits the slots from automobiles and sedans. In addition, it gives the value red to the color slot and mary to the owner slot. The unit joes.car inherits from automobiles and station.wagons. It gives the value blue to the color slot and joe to the owner slot.

A unit can inherit slots from multiple classes. Figure B.18 shows joes.car as a subclass of both station.wagons and fuel.injection.systems.

Rules are composed of two parts: antecedent (If clause), and consequent (Then clause). The rules can be run in a forward or backward direction, as described later in section on tapeworms. The If section of a rule contains expressions which monitor the slots in units. When the monitored slots change, the rule is triggered. In figure B.19 a forward rule is illustrated. The rule will trigger when the ignition.key slot and the ignition.system slot are both

```
The AUTOMOBILES Unit in AUTOSIM Knowledge Base

Unit: AUTOMOBILES in knowledge base AUTOSIM
Created by MULFORD on 10-29-87 15:29:25
Modified by root on 3-27-88 15:27:51
  Superclasses: ENTITIES in GENERICUNITS
  Subclasses: SEDANS, STATION.WAGONS, SPORTS.CARS
  Member of: CLASSES in GENERICUNITS

Member slot: COLOR from AUTOMOBILES
  Inheritance: OVERRIDE.VALUES
  Comment: "What color is this car?"
  Values: UNKNOWN

Member slot: NUMBER.OF.DOORS from AUTOMOBILES
  Inheritance: OVERIDE.VALUES
  Comment: "How many doors does this car have?"
  Value: UNKNOWN

Member slot: OWNER from AUTOMOBILES
  Inheritance: OVERRIDE.VALUE
  Comment: "Who owns this car?"
  Values: UNKNOWN
```

Figure B.16: Automotive Unit consisting of three slots.

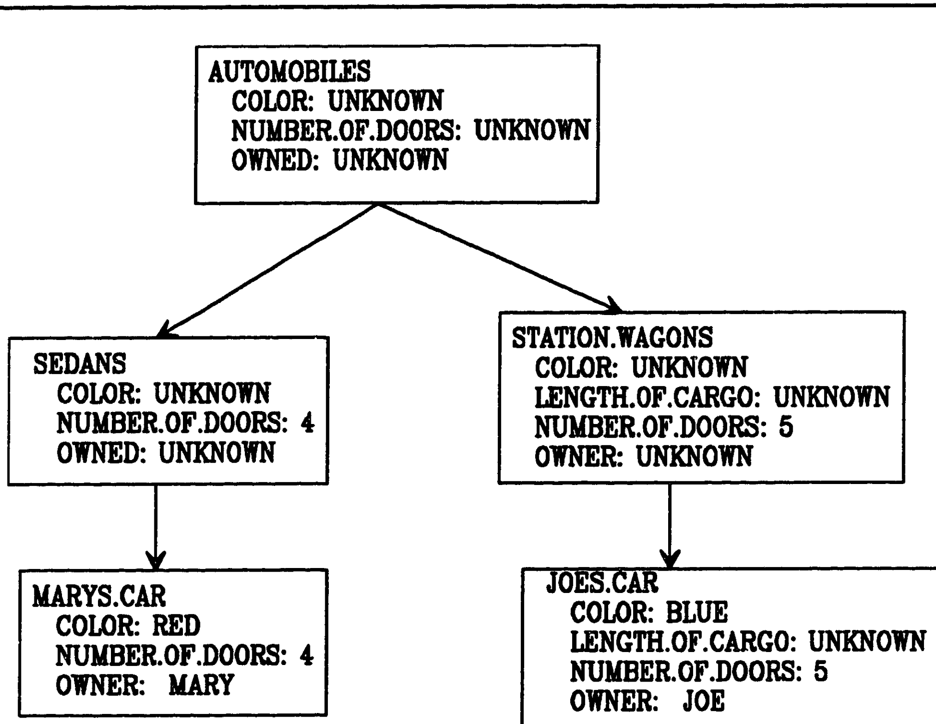


Figure B.17: The KEE class structure provides for the inheritance of slots.

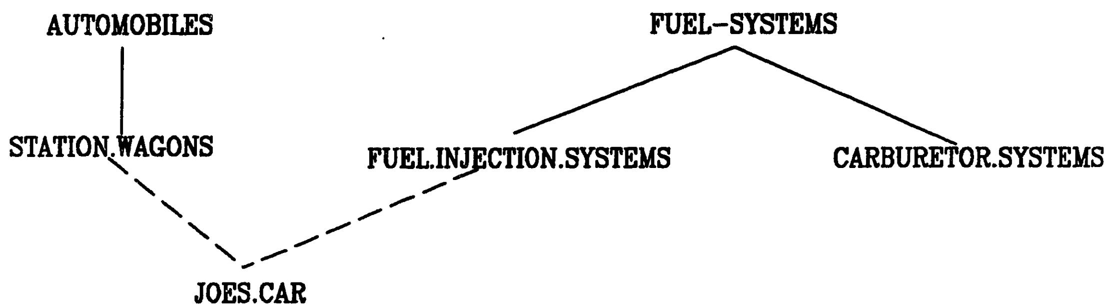


Figure B.18: KEE supports multiple inheritance. Joes.car inherits slots from station.wagons and fuel.injection.systems.

changed within the same unit. When the rule triggers the sparkplug.activity slot of the unit is changed to a value of firing.

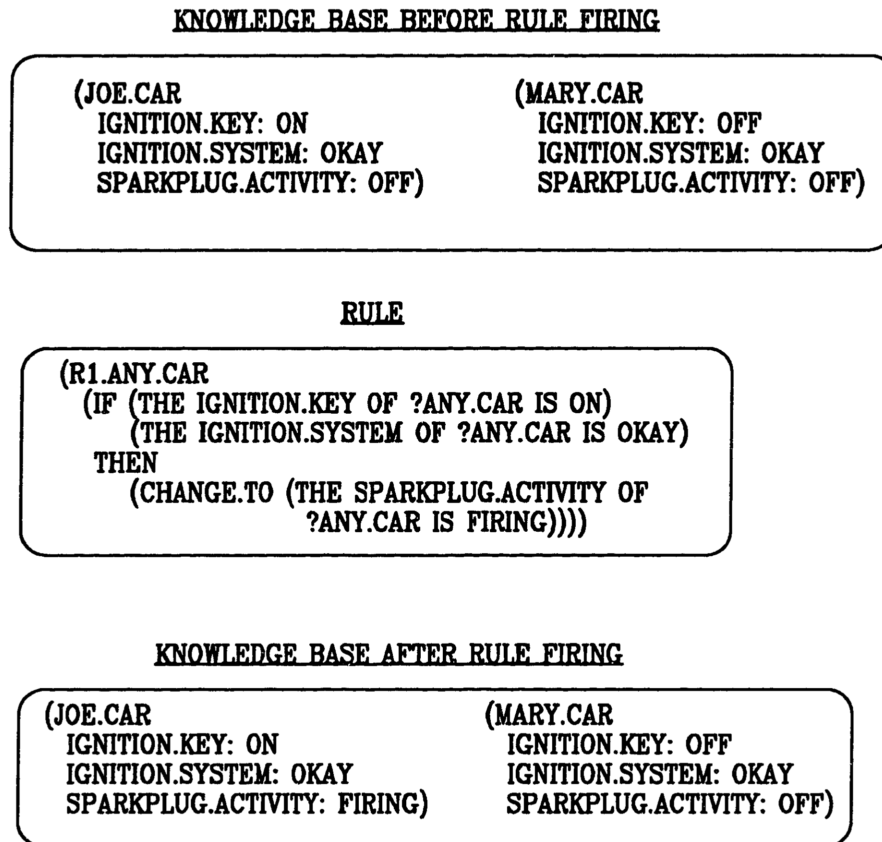


Figure B.19: The rule is fired when the ignition.key slot and ignition.system slot are modified for a unit. The rule sets the sparkplug.activity slot to firing.

Worlds [31] are KEE structures which can provide a context for units. Without worlds, all the units in a knowledge base are said to exist in the background. When the slots in a unit change, the unit in the background is changed. If the slots in a unit are changed in a world, then the changed unit exists in a world, and the unchanged unit exists in the background. Multiple worlds can exist. Figure B.20 shows four worlds: autowl, autow2, autow2.1, and autow2.2. Worlds autow2.1 and autow2.2 contain modifications of autow2. KEE rules can be restricted to the context of a world. They will only fire when changes are made to the world they are monitoring. When a rule fires it can create a new world.

Active Images supports the mapping of slots in KEE units to graphic images. When the values of the slot changes, the images change. Figure B.21 shows a collection of images which can be attached to slots in an automotive application.

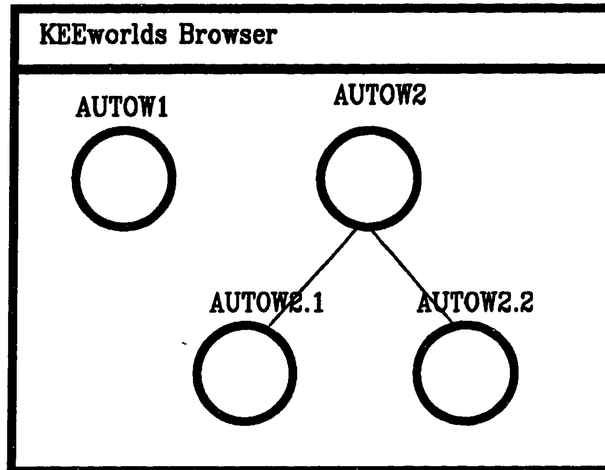


Figure B.20: In KEE, a world contains versions of a collection of units. Autow2.1 and autow2.2 contain modifications of autow2.

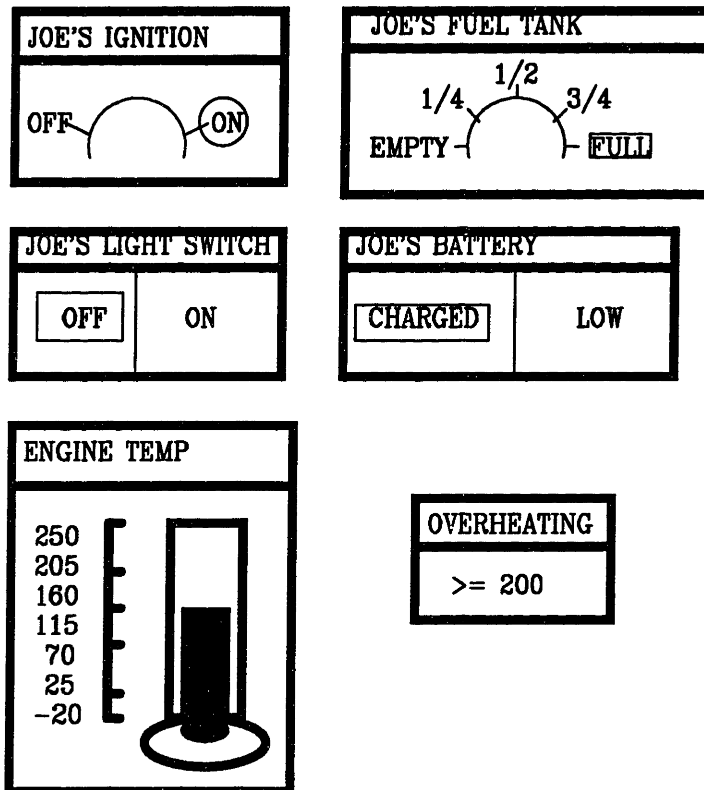


Figure B.21: KEE Active Images supports the connection of graphic images to unit slots. When the values of the slots change, the images change.

Distribution

KEE has no support for execution of distributed knowledge bases. Multiple knowledge bases can be used for storage of rules and units. All knowledge bases which are involved in an execution must be loaded together.

KEEconnection [4] is a subsystem which permits KEE to dynamically reference data stored in distributed relational databases. This subsystem consists of three modules, as follows:

1. Mapping module - describes how the relations in the databases are mapped onto units and slots in the knowledge base. The mapping does not have to be direct from relational columns into slots. A unit can consists of values from relations which have been *joined* together.

2. Translation - when the value of a slot is needed which references a relational database, the translation module dynamically retrieves the value using the SQL relational database language. When the value of a slot referencing a relational database is modified, the value is stored into the database.

3. Data communication - uses network facilities to transmit and receive data and queries from databases which can reside on different host computers.

Figure B.22 illustrates KEEconnection. The relational database contains the tables products and order-items. KEEconnection automatically creates the product and order-items units.

Name context within KEE is relative to a knowledge base A unit name must be unique within a knowledge base. A slot name must be unique within a unit. Versions of units can be created using worlds. The units must be unique within a world. Class provides a content but not name context. The names of units are unique within a knowledge base. A knowledge base can consist of multiple class structures. The classes specify slot inheritance type classes. A unit has two types of slots: member and own. Only member slots are inherited.

Action

KEE supports a wide variety of actions, some of which follow:

1. Messages can be sent to activate programs within unit slots.

2. Active values can invoke programs when slot values change.

3. Triggered rules can change slot values and invoke programs.

4. End-users using a menu interface can modify, query, and browse the knowledge base.

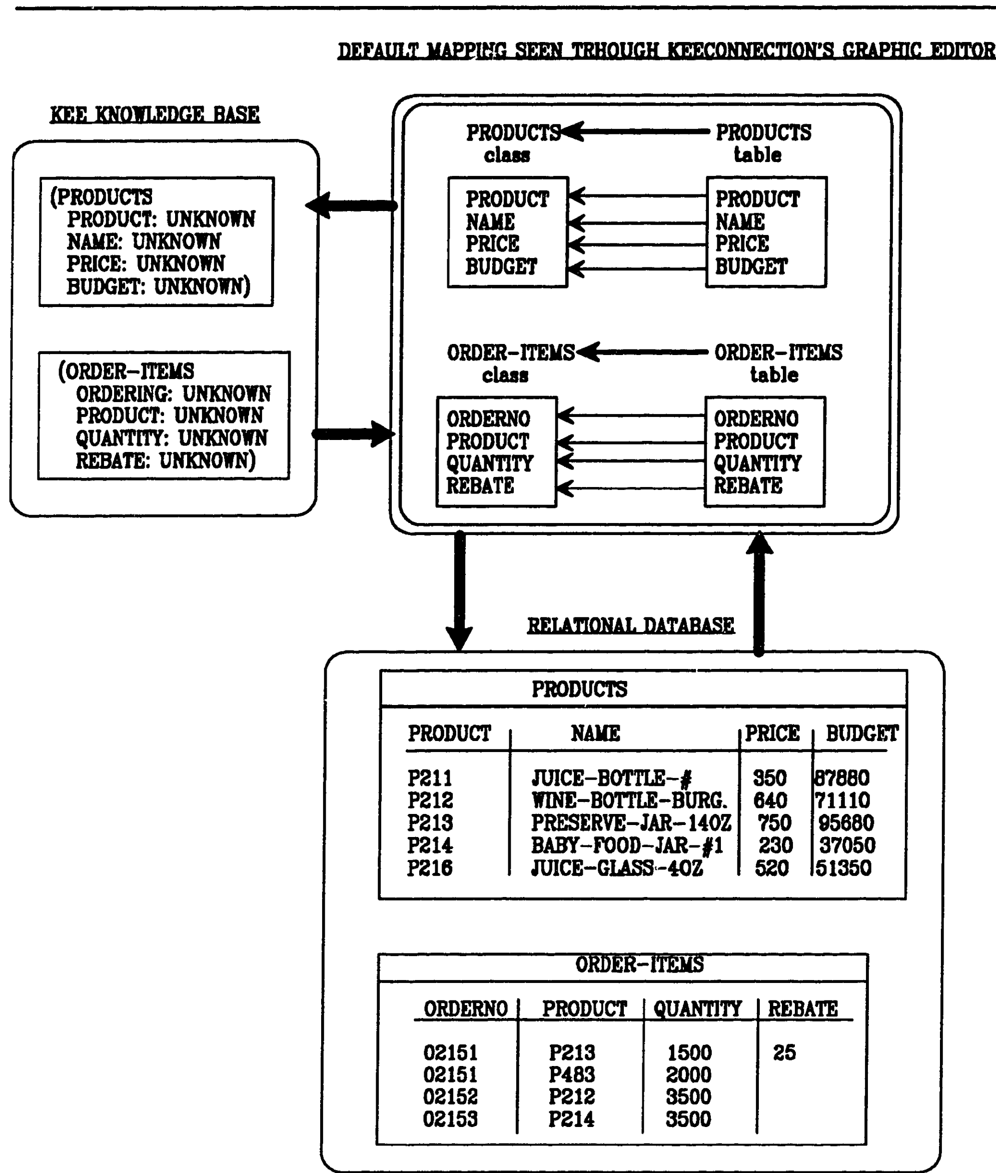


Figure B.22: KEEconnection provides for the automatic mapping of a relational database tables onto KEE units.

Action can be confined to a world. The truth maintenance system supports the retraction of action.

KEE does not directly support batch processing, parallel action, distributed action, action messages, or bureaucratic paths.

Tapeworms

KEE has two types of triggers: rules and active values. Rules trigger when a value in a slot changes. Active values trigger on the following conditions:

1. Adding an active value facet.
2. Referencing a value in a slot.
3. Changing a value in a slot.
4. Removing an active value facet.
5. Adding an assertion to a world.
6. Changing the focus to another world.
7. Changing the world inheritance flag. This flag specifies whether the world can be inherited.

Rules and active values are related to Ubik tapeworms as follows:

1. Commutativity - active values can be commutative or non-commutative. Rules are commutative.
2. Distribution - since there is no support for distributed knowledge bases in KEE, active values and rules cannot be placed on distributed knowledge bases and do not travel with messages.
3. Operations - active values and rules trigger on the modification of slot values. Rules running in the backward direction trigger on queries. Ubik tapeworms trigger only in the forward direction, but can trigger on query, insert, delete, sending of messages, and receiving of messages (or any combination of these operations), as shown in figure B.21.
4. Censoring - censors are provided by constraint facets, such as cardinality and value class. These restrict the values which can be placed within a slot. Censors are also provided by constraint rules. These are rules whose action are the issuing of false statements. A false statement issued within a world makes the world inconsistent. The truth maintenance system will make all the worlds which inherit from an inconsistent world also inconsistent. Constraint rules are weak censors, in that they don't prevent the inconsistent action from happening, but they prevent further action from occurring within the inconsistent worlds.

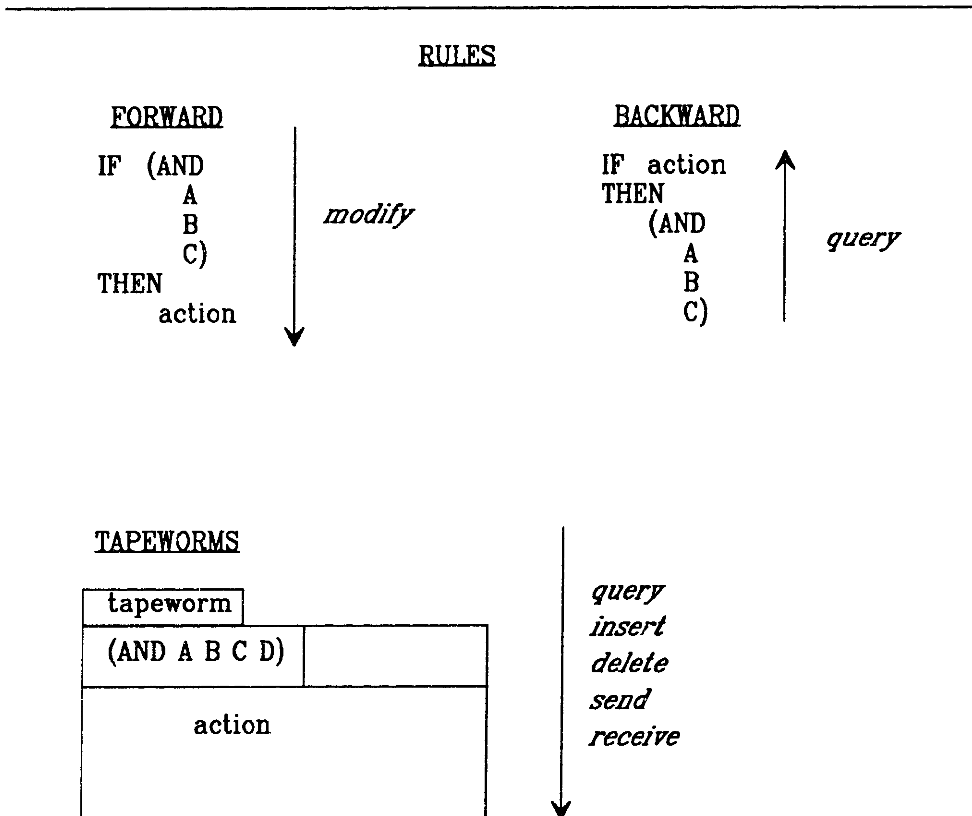


Figure B.23: KEE rules trigger on modification of slot values in the forward direction, and queries in the backward direction. Ubik tapeworms trigger only in the forward direction. They trigger on query, insert, delete, send, and receive (or any combination of these operations).

Questers

Rules running backwards in KEE provide a querying facility which works as follows:

1. KEE looks for units which match the query. If they are found, the query is completed; if not, processing continues, as discussed in 2.

2. KEE looks for rules whose consequence match the query. For the rules found, the antecedent of the rules are used to start a new subquery, and processing continues, as discussed in 1.

A query for forward rules can be devised by asserting a value within a world, and monitoring all the rules which fire as a result of the assertion. The truth maintenance system can be used to retract the assertion and undo the result of the assertion. This method might not completely work, because there are rules whose action cannot be reversed.

KEE provides browsers, which can display and navigate the unit's class and world structure. It also supplies active images, which display graphically the knowledge in the system.

KEE does not provide an ad-hoc query language which understands the structure of the KEE system, or the structure of the models built within the system. A query cannot examine the structure of the rules and what they reference. To find out how the rules and units are related, the user must cause the rules to be triggered and see the results, as described above. There is no support for tracing embedded units in slots.

KEEconnection provides for distributed queries, when the queried unit references a relational database.

Development

Not much development is possible within KEE. The class structure provides a tight connection between units. A change in the class structure would cause a change in all the units which inherit from the modified classes. This global propagation of changes ensures a system in which all the pieces are tightly tied together, making incremental changes difficult.

Ubik prototypes operate differently from classes. Ubik propagates changes to configurators created from prototypes at the time of a configurator's creation. After creation, the configurator is only loosely coupled to its prototype. During prototype development, a configurator can change the object it considers its prototype.

Regrouping is easily achieved in KEE. KEE's reasoning facility can create new units and slots from combinations of existing units and slots.

Power Relationships

The concept of cooperation and competition is not well defined in KEE. Worlds provide a notion of cooperation by permitting reasoning to simultaneously proceed in parallel worlds. Conflict can occur when worlds are merged. Consistency rules provide a means of weeding out inconsistent worlds. Multiple inheritance provides a form of competition in which conflicting facets can be inherited. An elaborate set of heuristics and procedures are provided to determine which facets take precedent when inheritance conflicts occur.

Focus of attention facilities are provided to schedule and dispatch rules. When an event occurs within a knowledge base, multiple rules can fire. These rules are scheduled using separate criteria for forward and backward rules. The forward rule schedule can be determined from the rule's priority and the complexity of the rule's antecedent clause. The backward rules can be specified as breadth-first or depth-first. A breadth-first rule is scheduled at the end of the dispatching queue; the depth-first rule is scheduled at the beginning of the dispatching queue.

Integration

KEE is a big system. All the pieces work together, but some are better integrated then others. For example, the world and truth maintenance system is well integrated with the rule system. This integration requires two types of rules: deduction rules and action rules. Additional commands are provided to restrict the triggering of a rule to the context of a world. A less-well integrated subsystem is the message system. The rule system is not aware of the message system, and cannot be triggered on the receipt or sending of a message.

B.4 Semantic Nets

A semantic net is a linked collection of objects. The objects in KEE are called units; they are linked together in an inheritance hierarchy. The objects in Ubik are called configurators. Ubik supports multiple kinds of links between the configurators. KEE and Ubik permit direct access to the implemented links; therefore, the behavior of the applications which use the semantic net is dependent on how they are implemented. Kl-one [13] is a semantic net system which takes a different approach. A Kl-one semantic net is composed of objects similar to units and configurators, but the implementation of the net cannot be directly referenced. Kl-one comes with a language which is used to describe, access, and change the net. Kl-one nets can be implemented in different ways, but the language interface ensures that all the implementations will produce the same behavior.

There are no defaults in a Kl-one net. Definitions represent solid unchanging facts about the world. Replacing definitions by defaults, according to Ron Brachman [14], would put the world on shaky pinnings by allowing the most obvious facts to be changed.

A complete application system cannot be built with Kl-one. Kl-one contains the concepts which make up an application, but it does not describe individual objects within the application or the application action.

Networks of Ubik configurators have the following properties, which are different than Kl-one nets:

- A Ubik network contains both objects and actions.
- The organization's structure and applications determine the position of an object in a taxonomy. A conceptual framework separate from the organization and its applications is not maintained.
- Organizational development incrementally changes the network to keep it relevant to the organizational action. The continual reorganization which occurs in organizations makes the Kl-one notion of fixed definitions impossible to maintain.
- The meaning of the network is in the use made of it. There is no abstract, application independent notion of meaning as in Kl-one.
- The implementation of the Ubik network can affect the action taken by the applications.

  In business organizations, action can also be dependent on the organization's implementation, as shown in the following example. The MIT purchase system has a subsystem for creating a purchase-order. Using this system, creation of a purchase-order usually takes a few days. A purchase-order can be created in one day if the purchase-requisition is hand-carried to the purchasing department. The purchasing department is a few blocks from my office at MIT. Rather than spend the

time getting all the approvals and hand-carrying the purchase order, I will usually purchase computer software by charging it to my personal credit card, and file for a reimbursement. If the implementation of the MIT purchasing system was changed, such that the purchase department was in the same building as my office, I would use purchase-orders for my software purchases.

Structure

A Kl-one net contains two type of concepts: primitive concepts, and defined concepts. A primitive concept is one which the Kl-one system cannot completely characterize. It has a name and attributes (called roles). The attributes form necessary conditions for a new concept to be subsumed by it. A defined concept is one that the Kl-one system can completely characterize. Its attributes form necessary and sufficient conditions for its identity. A newly defined concept which has the same attributes as an existing defined concept is identical with it. A defined concept need not have a name.

A Kl-one net is illustrated in figure B.24. At the top of a Kl-one net is one or more primitive concepts. For example, **person*** in figure B.24 is a primitive concept. A primitive concept is specified with an "*" after the concept's name. All other concepts in the net are subsumed by these top concepts. The subsumed concepts specify the difference between themselves and the concept to which they are attached. Difference can be specified as follows:

1. For a named primitive concept which inherits from the parent concepts, the difference is specified by the name of the concept and its position.
2. For a named primitive concept with roles which inherits from a parent concept, the difference is specified by the name of the concept, its position, and the roles. For example, from figure B.24, the following concepts differ from **person*** as follows:
   (a) the concept **man*** has role **sex**, value restriction **male** and cardinality a min and max of one.
   (b) the concept **parent*** has role **offspring**, value restriction **child** which is a subtype of person, and cardinality a min of one.
   (c) the concept **woman*** has role **sex**, value restriction **female**, and cardinality a min and max of one.
3. For a defined concept which inherits from the parent concepts, the difference is completely specified by its position. For example, from figure B.24, the following concepts differ from **man**, **parent**, and **woman**:
   (a) **father** inherits from **man*** and **parent***.
   (b) **mother** inherits from **parent*** and **woman***.

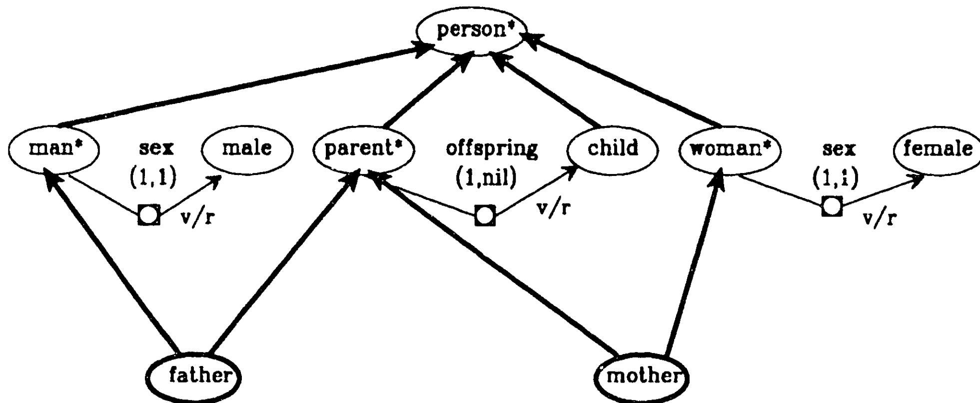


Figure B.24: A Kl-one net which specifies the terms which make up some family relationships.

In figure B.25 the Kl-one net is specified as a Ubik network. In Ubik, all concepts are primitive, roles are specified as prototype and part links, value restrictions specified by tapeworms.

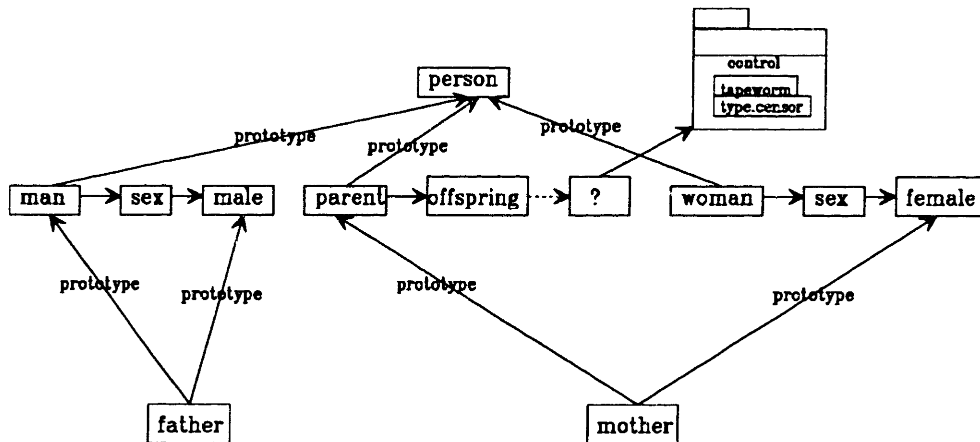


Figure B.25: A Ubik network which specifies the terms which make up some family relationships.

Distribution

There is no notion of distribution in Kl-one.

Context is provided by position not name. Each net has a name dictionary. Names are needed for primitive concepts. These names are global for a net. Defined concepts do not need a name, since they are completely defined by their position and roles. A name for a defined concept serves as

its documentation.

Action

A Kl-one net describes the conceptual structure of an application area. It doesn't state whether any individuals actually exist within the application. Kl-one has a primitive mechanism for describing individuals. More elaborate mechanisms have been designed in the Kl-two [63] and Krypton [12] systems. These systems use a Kl-one net for the description of the concepts, and a logic language for stating individuals, as shown in figure B.26. In this example, the logic expression states that two individuals exist, a father and mother. From the Kl-one net the system can deduce that a man, woman, parent, and person also exist.

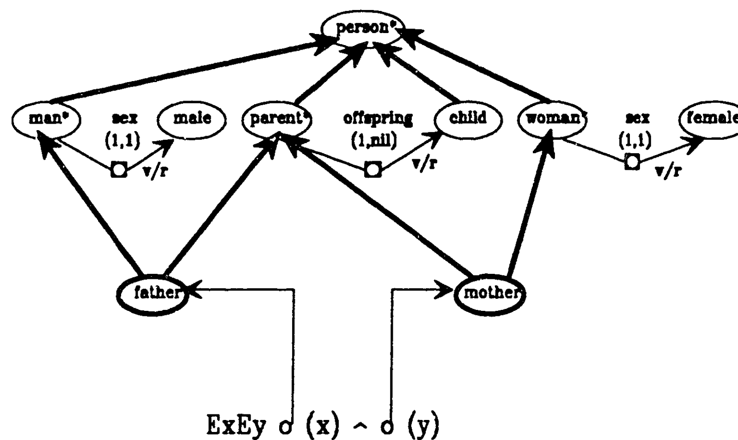


Figure B.26: Krypton and Kl-two have two components: the conceptual net language and assertional language. The conceptual net states the conceptual structure of the application; the assertional language specifies which individuals exist.

Action is not well defined within Kl-one. Applications using Kl-one interface with programs in one of the following ways: the application consist of multiple subsystems which have programs which access the Kl-one net, or the Kl-one net contains programs attached to the concepts which are invoked when the net is referenced or modified.

Tapeworms

Tapeworms in Kl-one are in the form of role restrictions and procedural attachment. Kl-one supports the following types of role restriction.

1. Value - the type of value which can be placed into a role.

2. Cardinality - the minimum and maximum number of values which can be placed in a role.

3. Inheritance role restriction - restricts the cardinality of inherited roles.

4. Inherited role differentiation - differentiates an inherited role into multiple roles.

5. Multiple role restriction - restriction between more than one role of the same concept. The Ubik tapeworm which restricts the salary of an employee to be less than his manager's is a restriction of this type.

Questers

The most fundamental operation of Kl-one is classification. Given a concept whose position is not known in the net, classification finds its position such that it is below all the descriptions which subsume it, and above all the descriptions which it subsumes. Figure B.27 shows a concept which inherits from `person*` and has a role `offspring` which is of type `child` and a role `sex` which is female. Classification, in the net shown in figure B.24, would find that this concept is the same as `mother`.

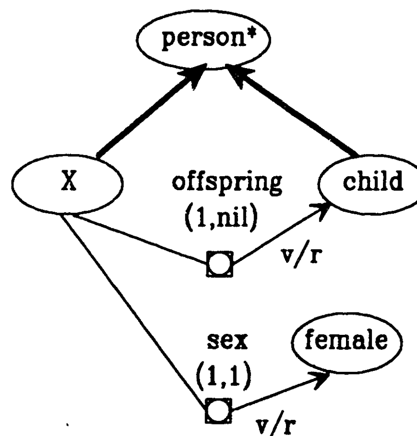


Figure B.27: The Kl-one classification operation would find that this concept is the same as `mother`, in the net in figure B.24.

Ubik does not contain a classification operation; if it did, the concept shown in figure B.28 would be classified as a mother in the Ubik network shown in B.25. Without classification, Ubik can not determine that it is a mother, but it can discover that it partially matches a parent and a woman.

Classification has many uses within Kl-one, some of which follow:

1. Incremental net building - In Ubik and KEE, the end-user would have to specify the placement of a new concept into a net; in Kl-one, the system can determine the placement from the structure of the net.

2. Query - Classification of an input concept would determine all the concepts which are equal to it, are subsumed by it, and it subsumes.

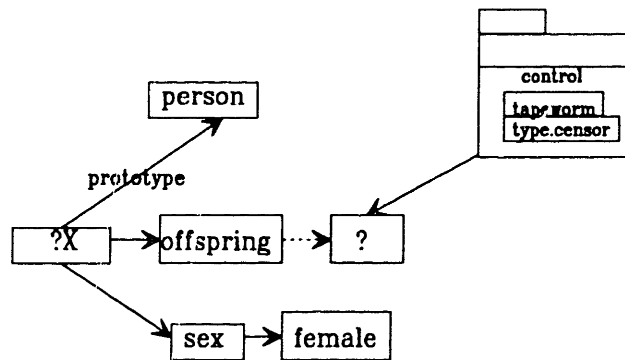


Figure B.28: This is the Ubik statement of Kl-one classification, as shown in figure B.27.

3. Search - Given a goal concept and an input concept, classification can discover if the input concept subsumes the goal. If it does, then the search can be restricted to the concepts in a net between the goal and the input concept.

Classification is a costly operation. Kl-one systems have taken two approaches in dealing with the complexity. The most frequent approach is to have a heuristic which can classify a concept. The heuristic approach can produce a sound classification—the placement conforms to the subsumption requirement; but the classification is not complete—the placement does not take in account all the possible placements. NIKL [6,56] represents a Kl-one system of this type. The other approach is to simplify the complexity of the Kl-one descriptions such that the classification is sound, complete, and can occur in polynomial time. Kandor [55] represents a Kl-one like system of this type.

Development

In the philosophy of Kl-one, a net represents permanent facts in the world. Other, more changeable concepts are represented in a different manner outside of Kl-one. Kl-one does not address development. If it did, development would entail modifying a part of the net, and then reclassifying all the other concepts in the net to relate them to the changed concept. This view of development would result in a change destablizing the whole conceptual structure of the organization.

Ubik views organizational development as a state in which the conceptual structure of the organization is continually changing. In Ubik, the net is defined by its current structure. A change in the structure would be confined to the specifically changed configurators. This local change could have significant effect on the action of the organization, but it would not

have a significant effect on the structure of the organization. An even more constrained change can be made using a tapeworm censor with replacement behavior. This change would not change the structure of the organization, but would partially change the action which the structure supports.

Power Relationships

Since distributed processing is not defined within Kl-one, the notion of power is not developed. Conflicts within Kl-one can result from multiple inheritance of roles. Kl-one has various heuristics to deal with this conflict.

Integration

Kl-one systems focus on the integration of defining and asserting concepts. Action in general is outside the scope of Kl-one, and thus not well integrated with the Kl-one system.

Bibliography

[1] *GoldWorks, Expert System Reference Manual.* GoldHill Computers Inc., Cambridge, Ma., 1987.

[2] *KEE Primer.* IntelliCorp, 1988.

[3] *KEE Version 3.1 User's Manual.* IntelliCorp, 1987.

[4] *KEEconnection: A Bridge Between Databases and Knowledge Bases.* IntelliCorp, 1987.

[5] *MIT Guide to Administrative Offices.* MIT Personnel Office, June 1984.

[6] *The NIKL manual.* Information Sciences Institute, University of Southern California, April 1986.

[7] *Paradox.* Borland International, Scotts Valley, Ca, 1988.

[8] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[9] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[10] Robert M. Akscyn, Donald L. McCracken, and Elise A. Yoder. KMS: a distributed hypermedia systems for managing knowledge in organizations. *Communications of the ACM*, 31(7):820–835, July 1988.

[11] L. W. Barsalou. Ad hoc categories. *Memory and Cognition*, 11:211–227, 1983.

[12] R. J. Brachman, R. E. Fikes, and H. J. Levesque. Krypton: a functional approach to knowledge r presentation. *IEEE Computer*, 16(10):67–73, 1983.

[13] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[14] Ronald J. Brachman. "I Lied about the Trees" Or, Defaults and Definitions in Knowledge Representation. *AI Magazine*, 6(3):80–93, 1985.

[15] John Brunner. *Shockwave Rider.* Harper and Row, 1975.

[16] Vannevar Bush. As we may think. *Atlantic Monthly*, 101–108, July 1945.

[17] Roy J. Byrd, S. E. Smith, and S. Peter de Jong. An actor-based programming system. In *Conference on Office Information Systems*, ACM SIGOA, June 1982.

[18] Susan Carey. *Conceptual Change in Childhood.* MIT Press, Cambridge, Massachusetts, 1985.

[19] Stanley Cavell. *Pursuits of Happiness, The Hollywood Comedy of Remarriage.* Harvard University Press, Cambridge, Massachusetts, 1981.

[20] W. D. Clinger. *Foundations of Actor Semantics.* AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[21] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[22] Jeff Conklin. Hypertext: an introduction and survey. *IEEE Computer*, 17–41, January 1988.

[23] Peter de Jong. Compilation into actors. *SIGPLAN notices*, October 1986.

[24] Peter de Jong. Ubik: a system for conceptual and organizational development. In W. Lamersdorf, editor, *Office Knowledge: Representation, Management, and Utilization*, North-Holland, 1988.

[25] Peter de Jong. The ubik configurator. In *ACM Conference on Office Information Systems*, March 1988.

[26] S. Peter de Jong. *The System Building System (SBS).* IBM Research Report RC 4486, IBM, 1973.

[27] S. Peter de Jong. The system for business automation - SBA: a unified application development system. In *Proceedings of the 1980 IFIP Congress*, IFIP, Tokyo, 1980.

[28] S. Peter de Jong and Roy J. Byrd. *Intelligent Forms Creation in the System for Business Automation.* IBM Research Report RC 8529, IBM, 1980.

[29] Philip K. Dick. *Ubik.* Doubleday, Garden City, New York, 1969.

[30] Steven Feiner. Seeing the forest for the trees: hierarchical display of hypertext structure. In *Conference on Office Information Systems*, pages 205–212, ACM SIGOIS and IEECS TC-OA, 1988.

[31] Robert E. Filman. Reasoning with worlds and truth maintenance in a knowledge-based system shell. *Communications of the ACM*, 31(4):382–401, April 1988.

[32] Mark E. Frisse. Searching for information in a hypertext medical handbook. *Communications of the ACM*, 31(7):880–886, July 1988.

[33] Jay Galbraith. *Designing Complex Organizations.* Addison-Wesley Publishing Company, 1973.

[34] David B. Guralnik, editor. *Webster's New World Dictionary.* William Collins + World Publishing Co. Inc., Cleveland, Ohio, second college edition, 1974.

[35] Frank G. Halasz. Reflections on notecards: seven issues for the next generation of hypermedia systems. *Communications of the ACM*, 31(7):836–852, July 1988.

[36] C. Hewitt, G. Attardi, and H. Lieberman. Specifying and proving properties of guardians for distributed systems. In *Proceedings of the Conference on Semantics of Concurrent Computation*, INRIA, Evian, France, July 1979.

[37] C. Hewitt and H. Baker. Laws for communicating parallel processes. In *1977 IFIP Congress Proceedings*, IFIP, 1977.

[38] Carl Hewitt. The challenge of open systems. *Byte*, 10(4):223–242, April 1985.

[39] Carl Hewitt. Offices are open systems. *ACM Transactions on Office Information Systems*, 4(3):271–287, July 1986.

[40] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[41] Carl Hewitt and Peter de Jong. Analyzing the roles of descriptions and actions in open systems. In *Proceedings of the National Conference on Artificial Intelligence*, AAAI, August 1983.

[42] Carl Hewitt and Peter de Jong. Open systems. In M. L. Brodie, J. L. Mylopoulos, and J. W. Schmidt, editors, *Perspectives on Conceptual Modeling*, Springer-Verlag, 1983.

[43] Carl E. Hewitt. The apiary network architecture for knowledgeable systems. In *Conference Record of the 1980 Lisp Conference*, Stanford University, Stanford, California, August 1980.

[44] W. Kornfeld. *Concepts in Parallel Problem Solving.* PhD thesis, Massachusetts Institute of Technology, 1982.

[45] W. A. Kornfeld and C. Hewitt. The scientific community metaphor. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1), January 1981.

[46] George Lakoff. *Women, Fire, and Dangerous Things.* The University of Chicago Press, 1987.

[47] Corneilis J. Lammers and David J.Hickson. A cross-national and cross-institutional typology of organizations. In Corneilis J. Lammers and David J.Hickson, editors, *Organizations Alike and Unlike*, pages 420–434, Routledge and Kegan Paul, London, 1979.

[48] H. Lieberman. *A Preview of Act-1.* A.I. Memo 625, MIT Artificial Intelligence Laboratory, 1981.

[49] H. Lieberman. *Thinking About Lots of Things At Once Without Getting Confused: Parallelism in Act-1.* A.I. Memo 626, MIT Artificial Intelligence Laboratory, 1981.

[50] David G. Lowe. Cooperative structuring of information: the representation of reasoning and debate. *International Journal of Man-Machine Studies*, 23:97–111, 1985.

[51] Carl Manning. *Acore: The Design of a Core Actor Language and its Compiler.* Master's thesis, Massachusetts Institute of Technology, 1987.

[52] James G. March and Herbert A. Simon. *Organizations.* John Wiley and Sons, Inc., New York, 1958.

[53] Marvin Minsky. *The Society of Mind.* Simon and Schuster, 1986.

[54] G. L. Murphy and D. L. Medin. The role of theories in conceptual coherence. *Psychological Review*, 92(3):289–316, 1985.

[55] Peter F. Patel-Schneider. Small can be beautiful. In *Proceedings IEEE Workshop on Principles of Knowledge Based Systems*, October 1984.

[56] Peter F. Patel-Schneider. Undecidability of subsumption. *Artificial Intelligence*, 39(2), June 1989.

[57] Teresa L. Petramala. Don't try to tell me that the check is in the mail. *New York Times*, March 1989. Weekend Advance 3/24/89, Copyright 1989 The New York Times.

[58] Hilary Putnam. The meaning of 'meaning'. In H. Putnam, editor, *Mind, Language, and Reality*, pages 215–271, Cambridge University Press, New York, 1975.

[59] W. Richard Scott. *Organizations.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[60] Ehud Shapiro. *Concurrent Prolog*, chapter A Subset of Concurrent Prolog and Its Interpreter. The MIT Press, Cambridge, Ma., 1987.

[61] Herbert A. Simon. On the concept of organizational goal. *Administrative Science Quarterly*, 9:1–22, June 1964.

[62] A. Tversky. Features and similarity. *Psychological Review*, 84:327–352, 1977.

[63] Marc Vilain. The restricted language architecture of a hybrid representation system. In *IJCAI-85*, pages 547–551, 1985.

[64] Max Weber. *The Theory of Social and Economic Organization.* The Free Press, New York, 1947.

[65] Nicole Yankelovich, Bernard J. Haan, Norman K. Meyrowitz, and Steven M. Drucker. Intermedia: the concept and the construction of a seamless information environment. *IEEE Computer*, 81–96, January 1988.

[66] M. M. Zloof. Query by example. In *Proceedings of the NCC*, pages 431–438, AFIPS, 1975.

[67] M. M. Zloof and S. Peter de Jong. The system for business automation SBA: programming language. *Communications of the ACM*, 20(6), June 1977.