TOWARDS A COORDINATION COOKBOOK: RECIPES FOR MULTI-AGENT ACTION

by

KEVIN GHEN CROWSTON

A. B. Applied Mathematics (Computer Science)
Harvard University
(1984)

Submitted to the Alfred P. Sloan School of
Management in partial fulfillment of the
requirements for the degree of

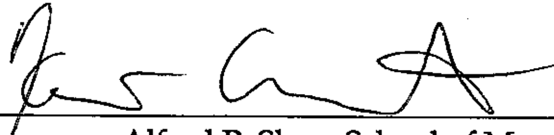
DOCTOR OF PHILOSOPHY IN MANAGEMENT

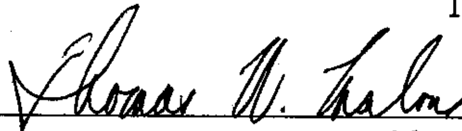
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1991

© Massachusetts Institute of Technology 1991
All rights reserved

Signature of author 
Alfred P. Sloan School of Management
18 January 1991

Certified by 
Thomas W. Malone
Patrick J. McGovern Professor of Information Systems
Thesis Supervisor

Accepted by ______________________________
James Orlin, Chairman
Doctoral Program Committee
Alfred P. Sloan School of Management

ABSTRACT

This thesis presents the first steps towards a theory of coordination in the form of what I call a *coordination cookbook*. My goal in this research is hypothesis generation rather than hypothesis testing: I attempt to develop a theory of coordination grounded in detailed empirical observation. I am especially interested in using this theory to identify ways of coordinating that may become more desirable when information technology is used to perform some of the coordination.

I address the following question: how can we represent what people do to coordinate their actions when they work together on common goals, in a way that reveals alternative approaches to achieving those goals? To answer this question, I study groups of people making engineering changes to complex products as an example of a coordination-intensive task. I perform detailed case studies of the change process in three organizations: an automobile manufacturer, a commercial aircraft manufacturer and a computer system software developer.

To analyze these cases, I develop a technique for describing the behaviour of the members of an organization, based on research in distributed artificial intelligence (DAI). I first develop a data-flow model of the change process to identify what information was used and how it was processed by the different members of the organization. Then, using ideas from DAI, I model what each individual must have known about the task and the rest of the organization to act as observed.

To develop a theory of coordination, I generalize from these specific individuals to the kinds of tasks they performed. I develop a typology of interdependencies between organizational tasks and objects in the world (including resources and products). This typology includes four categories of coordination needs, due to interdependencies between: (1) different tasks, (2) tasks and subtasks, (3) tasks and objects in the world and (4) different objects.

I then re-examine the cases to identify the coordination methods used to address these needs. (These coordination methods are similar in spirit to the weak problem solving methods of cognitive science.) I represent each method by a set of what I call *coordination recipes* that identify the goals, capabilities and knowledge of the individuals involved. In some cases, consideration of the possible distributions of these elements suggests approaches other than those actually observed. This framework allows an analyst to abstract from a description of how a particular organization performs a task to a description of the coordination needs of that task and a set of alternative coordination methods that could be used to address those needs.

The results of my thesis should be useful in several ways. A better understanding of how individuals work together may provide a more principled approach for designing new computer applications, for analyzing the way organizations are currently coordinated and for explaining perceived problems with existing approaches to coordination. By systematically exploring the space of possible coordination strategies, we may be able to discover new kinds of organizations—organizations in which humans and computers work together in as yet unimagined ways.

Thesis supervisor: Thomas W. Malone
Patrick J. McGovern Professor of Information Systems

Thesis committee: John S. Carroll
Deborah Ancona

PREFACE

O what a beautiful morning at the end of November, in the beginning was the word, sing to me, goddess, the son of Peleus, Achilles, now is the winter of our discontent. Period, new paragraph.

—Eco, *Focault's Pendulum*

As I was cleaning out my office the other day, I came across a Sloan School personnel list from the spring of 1984, shortly before I entered the Ph. D. program. When I compared it to the most recent personnel list from the fall of 1990, I noticed many changes. A couple of changes in the form of the list struck me as being particular relevant to my study of information technology.

First, while the old list had been typed by hand, the new one was prepared on a word processor. The new list had no typographical errors or crossed out lines like the old one did. This kind of immediate improvement in individual performance is one reason information technologies like word processors have become so widely used. (It is certainly the case that without a word processor, this thesis might never have been finished.)

However, a more subtle change was also visible. Where the old list said simply, "Spring 1984", the new list showed the exact date and the time it had been printed. The new technology had reduced the cost of preparing a personnel list so far that several lists might be prepared in a single semester or even on the same day. This kind of second-order effect will, in the long run, have a much more pronounced effect on the way organizations do business, but these kinds of changes are much harder to predict.

Second, where the old list gave only the name, room and phone number, the new list added electronic mail addresses. Only about 70% of the people on

the last had one (or had it listed), but still, in the last seven years, electronic mail had gone from something only a few computer scientists have to a medium used regularly by otherwise non-technical academics. It is now usual for a conference registration form or other mailing list to request electronic mail addresses. A similar change has taken place in regular business circles; within the last few years electronic mail has gone from unheard of to commonplace[1], at least within large firms.

As a first-order effect, electronic mail makes communication cheaper and faster, especially with correspondents in other cities, thus permitting greater interaction between geographically separated individuals. However, as the presence of the addresses on an internally distributed list shows, electronic mail may be used even for communication within a single site, between people who could (and probably do) talk face-to-face. The second-order effects of this cheaper communication are only beginning to be seen.

The widespread use of electronic mail is especially interesting to me because it is an example of an information system that supports not just individuals, but groups. (Individual electronic mail is something of an oxymoron.) One goal of my thesis is to suggest ways to determine what kinds of group support systems organizations might find useful and to predict the first and second order changes associated with their use.

* * *

A comparison of the contents of the two lists revealed many other changes over the last seven years. Of the 234 names on the first list and 276 on the second, only 96 were the same, meaning, I suppose, that I have had the opportunity to work with at least 414 faculty and staff (not counting those who arrived and left again within seven years). Other groups have turned over even more thoroughly: of the 90 or so current Ph. D. students, only 6 have been here

---

1 Based on informal surveys of attendees at talks; audience members were asked to raise their hands if they use electronic mail. (Malone, personal communication.)

since 1984 (proving, I suppose, that graduate students really do not last longer than the faculty, even if it sometimes seems that way).

Continuing in this quantitatively reflective mood, I have determined that outside of school, in the same seven years, I have had no fewer than thirty-five housemates. I can not even begin to estimate the number of friends I've made—but the set of my immediate family members at least has remained unchanged.

I owe many of these people for many things over the last seven years. It's unfortunate that I didn't keep a diary; my public thanks will have to rely instead on retrospection, a notoriously unreliable data source (see Chapter 3).

I would like to thank:

First, my advisor, Tom Malone, for doing all the things advisors are supposed to do and more—listening to and discussing my ideas, reading drafts and suggesting changes, believing that I would actually finish and paying me until I did and generally getting me started in an academic career—and all for a student who doesn't (didn't?) really believe in working hard. The fact that neither one of us can remember who thought of some of these ideas first is a good indication of the collaborative nature of our work.

The other members of my thesis committee, Deborah Ancona and John Carroll, for patiently listening to ideas about organizations from a rather different field (left field?) and for their many suggestions for improving their presentation.

My co-researchers: Felix Lin, who participated in an earlier field study, documented in (Crowston, et al., 1987), that laid the foundation for this study; and Stephen Brobst, who arranged and participated in the field work at my first case site, Computer Systems Co.

The employees of my three case sites, Computer Systems Co., Car Co. and Airplanes, Inc. for generously taking the time to answer my questions and tell me "what they did when they did engineering changes". I hope they will recognize themselves (collectively) in my cases and find something new in my analysis.

The Sloan Information Technologies faculty, for providing such a comfortable environment in which to work, and Wanda Orlikowski in particular for several times helping me figure out what it was I was trying to say.

Randy Davis and Lynn Stein for providing a few real insights from the world of artificial intelligence and common sense reasoning.

The students and staff of the Center for Coordination Science—Jin Lee, David Rosenblitt, Paul Resnick, Kum-Yew Lai and Mark Ackerman—and other Sloan Ph. D. students, present and past, including (at least) Andy Trice, M. Bensaou, David Robertson, Yannis Bakos and Michael Epstein, all of whom have greatly contributed to my intellectual quality of life. Talking with them and especially with Brian Pentland, my co-coordinationist, have helped me clarify my thinking and, as importantly, made me realize that not knowing what I was doing is a stage everyone goes through.

Judy Shapiro, Lisa Fleischer and Allen Feinstein, who read and instructively failed to understand various formerly poorly written bits of this document.

My housemates, for making it worth going home.

And finally, my family, who most recently tolerated my rewriting my thesis during our Christmas vacation and especially my sister Clare, who lent me her laptop computer with which to rewrite it; truly a case of information technology making the possible easy and the otherwise impossible only time-consuming.

—Paris, 9 January 1991

TABLE OF CONTENTS

Abstract 3
Preface 5
List of Tables 15
List of Figures 17

1 Towards a coordination cookbook 19
- 1 Coordination 20
  - 1.1 Some properties of coordination 21
  - 1.2 Elements of coordinated situations 25
- 2 Why a cookbook? 28
  - 2.1 Uses of a cookbook 29
- 3 Overview of the thesis 31
  - 3.1 Modelling coordination 31
  - 3.2 Case studies 31
  - 3.3 Typology of coordination methods 32
  - 3.4 Contributions 33
- 4 Related work 33
  - 4.1 Coordination in organizational behaviour 35
  - 4.2 Coordination in artificial intelligence 40
  - 4.3 Conclusion 41

2 Modelling coordination 45
- 1 Why model? 45
- 2 What to model 47
  - 2.1 Information processing view of organizations 48
  - 2.2 Simplifying assumptions 49
- 3 My approach to modelling coordination processes 51
  - 3.1 A brief introduction to logic 52
- 4 Modelling knowledge about the world 54
  - 4.1 Assumptions about knowledge 56

4.2 Organizational knowledge ... 58
5 Modelling actors' goals ... 59
5.1 Where do goals come from? ... 61
6 Modelling actions ... 62
6.1 Preconditions ... 62
6.2 The frame problem ... 63
6.3 Primitive actions ... 64
6.4 Complex actions ... 67
6.5 Where do capabilities come from? ... 68
7 Modelling communication between actors ... 69
7.1 Declarative speech acts ... 70
7.2 Imperative speech acts ... 71
7.3 Interrogative speech acts ... 72
7.4 Performative speech acts ... 73

3 Study design ... 75
1 Case study methodology ... 75
2 Task selection ... 77
2.1 About engineering changes ... 78
2.2 Parts ... 79
2.3 Stages in implementing engineering changes ... 80
2.4 Engineering change control ... 80
2.5 Why study change management? ... 82
3 Case site selection ... 83
3.1 Brief summary of sites ... 86
4 Data collection ... 87
4.1 Data collected ... 88
4.2 Validation of data ... 90
5 Analysis technique ... 91
5.1 Information-flow models ... 91
5.2 Intentional models ... 96
3-1 Interview outline ... 103
3-2 Sample transcript ... 105
3-3 Sample information flow model ... 113
3-4 Sample intentional model ... 113

4 Computer Systems Co. ... 121
1 Overview of site ... 121
1.1 Data collection ... 122
1.3 Characteristics of the product ... 122
1.2 Characteristics of the organization ... 127

2 The software development process ... 128
3 The software change process ... 133
3.1 Reasons for changes ... 133
3.2 Goals of the change process ... 133
3.3 Overview of process ... 134
3.4 An informal description of the change process ... 135
3.5 Perceived problems with the change process ... 145
3.6 Reorganization ... 147
4 An information-flow model of the change process ... 148
4.1 Overview of model ... 148
4.2 Components of the model ... 148
4.3 Determining a change is necessary ... 150
4.4 Solution development and approval ... 154
4.5 Solution implementation ... 156
5 Intentional model ... 157
4-1 Information-flow model ... 159
4-2 Intentional model ... 167

5 Site B: Car Co. ... 173
1 Overview of the site ... 173
1.1 Data collection ... 173
1.2 Characteristics of the product ... 174
1.3 Characteristics of the organization ... 177
2 The engineering development process ... 181
2.1 Concept development ... 182
2.2 System definition ... 182
2.3 Design and development ... 183
2.4 Design validation ... 185
2.5 Product validation ... 186
2.6 Production ... 187
2.7 Simultaneous engineering ... 187
3 The engineering change process ... 188
3.1 Reasons for changes ... 189
3.2 Changes at different phases ... 191
3.3 Description of the change process ... 192
3.4 Why are changes difficult? ... 199
3.5 Perceived problems with the change process ... 199
4 An information-flow model of the change process ... 201
5 Intentional model of the change process ... 201
5-1 Information-flow model ... 203
5-2 Intentional model ... 221

6 **Site C: Airplanes, Inc.** **227**
1 Overview of site 227
1.1 Data collection 227
1.2 Characteristics of the product 228
1.3 Characteristics of the organization 232
2 The engineering development process 233
2.1 New airplane program 234
2.2 Overview of production process 236
3 The change process 239
3.1 Dimensions of changes 240
3.2 Goals of the change process 240
3.3 An informal description of the change process 240
3.4 Perceived problems with the change process 253
4 Models of the change process 253
6-1 **Information-flow model** **257**
6-2 **Intentional model** **271**
7 **Recipes for multi-agent action** **279**
1 A typology of coordination problems and methods 279
1.1 Objects 285
1.2 Tasks 286
2 Recipes 287
2.1 Sources of recipes 288
3 Category 1a: Dependencies between tasks and subtasks 289
3.1 Task decomposition 289
3.2 Supertask induction 291
4 Category 1b: Dependencies between different tasks 291
4.1 Avoiding constraints 294
4.2 Conflicting creation of objects 295
4.3 Output of one task is input of another 298
4.4 Conflicting use of objects 302
5 Category 2: Dependencies between tasks and objects 303
5.1 Identifying task requirements 305
5.2 Identifying actors who can perform the task 306
5.3 Choosing an actor to perform the task 311
5.4 Getting a chosen actor to perform a task 312
5.5 Assigning resources 314
5.6 Assigning coordination tasks 315
6 Category 3: Dependencies between objects 317
6.1 Identifying interdependencies 318
6.2 Evaluating effect of interdependencies 319

7 Related work 320
7.1 Interdependence in organizational behaviour 320
7.2 Dependence in AI research 323
7.3 Conclusion 325

8 Conclusion 327
1 Summary of results 327
2 Solutions to benchmark problems 329
2.1 Organizational redesign 330
2.2 Designing computer-support systems 333
2.3 Effects of intensive use of information technology 334
3 Future work 335
3.1 Where do goals come from 335
3.2 Development of coordination knowledge 335
3.3 Computer simulations 336
3.4 Presentation of the models 336
4 Conclusion 338

References 341

LIST OF TABLES

2.1 Processes underlying coordination 66
4.1 Actor types for the Computer System Co. model 150
4.2 Messages for Computer Systems Co. information flow model 151
5.1 Rate of changes and cost and number of parts involved in changes at different stages of design 190
5.2 Actor types for the Car Co. model 201
6.1 Actors types for the Airplane Inc. model 254
7.1 Typology of dependencies between tasks and objects 280
7.2 Typology of coordination needs and corresponding chapter section 281
7.3 Coordination recipes 282
7.4 Examples of objects classified by shareability and reusability 285
7.5 Typology of coordination problems by operations and type of common object 293

List of Figures

4.1 Organizational structure of Computer Systems Co. Software division 129
4.2 Overview of change process for Computer Systems Co. 134
4.3 Overview of information-flow model for Computer Systems Co. 149
5.1 Organizational structure of Car Co. Engineering and Manufacturing 178
8.1 A collection of general purpose (circles) and specialized actors (triangles and squares) 336
8.2 Alternative ways of performing a task: with or without task decomposition and with no, hierarchical or market task assignment 337
8.3 Organizational structures resulting from different combinations of the abstract steps above 338

1

TOWARDS A COORDINATION COOKBOOK

"Begin at the beginning," the Kind said, very gravely, "and go on till you come to the end: then stop."

—Lewis Carroll, *Alice in Wonderland*

What good are organizations? Obviously, organizations exist for many reasons. One of the most important is to channel the efforts of the organization's members in ways that allow the organization to accomplish things that no individual working alone could do. For example, an automobile is such a complex product that no one individual can be said to know how to design and build one, yet several organizations routinely do exactly that.

This ability has a cost: much of the work done by the employees of an automobile company has little to do with actually building automobiles (what I call *production* work). Instead, these workers spend their time *coordinating* their actions and the actions of others. As yet, however, we have only an informal understanding of what coordination work is or how it is useful. This lack of understanding limits our ability to diagnose problems with existing organizations or to imagine new ways to organize in response to changing demands from the environment or abilities of the organizations.

This thesis presents the first steps towards a theory of coordination in the form of what I call a coordination cookbook. My goal in this research has not been hypothesis testing; rather, I have attempted to develop a grounded theory of coordination. I addressed the following research question: how can we represent what people do to coordinate their actions when they work in groups to achieve common goals, in a way that reveals alternative approaches

to achieving those goals? I am especially interested in identifying ways of coordinating that may become more desirable when information technology is used to perform some of the coordination.

To make these issues more concrete, I propose three "benchmark" problems as a test of the theory to be developed. First, in what ways can a given organization be arranged differently while achieving the same goals? Second, what kinds of information technologies will be useful to support a particular organization? Finally, how might the preferred structure for an organization change with extensive use of information technology?

My approach to answering these questions has two parts. First, I have developed a technique for describing the members of an organization, based on research in artificial intelligence. Second, I have developed a typology of coordination problems and alternative ways those problems can be addressed, based on an empirical study of three organizations. This framework allows me to abstract from a description of how a particular organization performs a task to a description of the coordination needs of that task and then to a set of alternative coordination methods that could be used to address those needs. In Chapter 8, I will discuss in more detail how the theory developed in this thesis helps address these problems.

1 Coordination

To start, let me informally define coordination. We all have an intuitive notion of what coordination is. For example, when we see a smoothly running factory or a skillful basketball team, we might say that the workers or team members are well coordinated. Three examples from one case site will illustrate the kinds of behaviours I hope to explain with my theory.

1) Customers with problems call the response center to report them. The response center first looks for solutions to the problems in a database of known problems.

2) If the solution is not found in the database, the reported problems are routed, in a several-step process, to an engineer responsible for the module with the problem to be fixed.

3) The responsible engineer attempts to fix the problem. The changes considered may involve multiple modules; the software engineer determines which other modules may be affected, using mental and sometime physical models of the system and consults with the engineers responsible for those modules. This process may result in those other engineers making additional changes to their modules.

Our intuitive definition of the word "coordination" may lead us into difficulties, however: in particular, we seem to use the word in at least two different ways, similar to the way we use "diet" to mean both what people eat in general and a specific regime designed to achieve some goal, such as losing weight. The general sense of "coordinated" is that the individuals actually achieve their common goals; the specific sense is that the individual actors work together in a closely coupled manner. The first meaning is usually seen as a positive outcome; the second is relatively neutral. In order to separate these meanings, I will replace uses of "coordinated" in the first sense by some notion of group performance. I will then focus on the ways people "coordinate" in the second sense.

It is worth noting, for example, that one possible way of being coordinated (in the first sense) is by not being coordinated at all (in the second sense), that is, people may succeed in achieving their common goals at low cost even though the individuals make no special effort to work together. In general, of course, additional efforts are necessary.

1.1 Some properties of coordination

Even though my initial definition of coordination is essentially, "I'll know it when I see it," I can still identify a number of what seem to be important characteristics.

Coordination is an information processing task. An important part of what goes on in coordination involves communication and processing of

information. The organizational structure of the group, by constraining the possible patterns of communication, has a powerful effect on the way a group is coordinated. In fact, as Thompson (1967) argues, organizational structures exist mostly to enhance particular patterns of coordination. Note that the communication necessary need not be coincident with the task in question: one way of being coordinated is to plan everything in advance so no communication is necessary at the time the task is performed. Another way is to do nothing in advance and communicate only as required by the situation. Different levels of communication may be handled differently; for example, actors might communicate in advance to decide what kind of information is needed and communicate only that information as necessary during the process.

Although this dissertation is focused on understanding human organizations, I believe that the same issues are relevant to the coordination of any goal-directed system. Examining these issues from many points of view may provide additional insight into each domain. Fox (1981), for example, suggested using ideas from organization theory to analyze distributed systems. In this thesis I will attempt the opposite, that is, use ideas developed in computer science to aid in the understanding of human organizations.

Coordination depends on the goals of the organization. What is considered coordination, production or wasted effort depends on the goals of the group being analyzed. For example, in a car company, accounting would likely be considered a coordination task, since balancing the books does not directly help build cars but rather tracks the allocation of resources. For an accounting company, however, "book balancing" may be a primary goal and accounting therefore a production task. If an individual performs some tasks which do not help the organization reach its goals in any way, those tasks may be considered wasted effort. For example, designing a part which will not be used would probably be considered a waste of time. In some cases, it may seem that individuals are doing unnecessary things, like chatting in the hallways, but those actions may be necessary to achieve other goals, like

maintaining communications channels, individual motivation or the social cohesiveness of the organization.

Coordination is attributed to a situation by observers. Attributing goals to groups is problematic. As Cyert and March (1963) pointed out, "people have goals; collectivities of people do not" (p. 26). Although this statement is true in a strict sense, we will find it useful to treat organizations as if they had goals. As discussed in Malone and Crowston (1990), the actors involved in a situation may or may not all agree on the identification of the goals or the boundaries of the organization. Instead, one or more of these components may be attributed by an observer in order to analyze the situation in terms of coordination. For instance, in analyzing market, we might as observers regard the goal to be achieved as one of optimally allocating resources to maximize consumer utilities (e.g., Debreu, 1959). Even though no single individual in the market necessarily has this goal, observers might evaluate market coordination in terms of how well it achieved this goal.

In taking this approach, we adopt Dennet's (1987) intentional stance: since there is no completely reliable way to determine someone's goals (or if indeed they have goals at all), we, as observers, can only impute goals to the actors. The key issue then is how useful the imputed goals are for understanding the behaviour of the system. Therefore, in order to analyze how the actions of members of an organization are coordinated, we will assume that they are attempting to perform some task. Other researchers have found this focus useful; Hackman (1969), for example, pointed out that tasks are important to understanding behaviour. Organizations are often created explicitly to perform some task. I will therefore assume the actors are attempting to perform a particular task and analyze how they do it.

Once the organization has reached a steady state, this is not an unreasonable assumption. Cyert and March (1963) suggested a mechanism of bargaining and side payments through which individuals reach a consensus on common goals, but note further that, in fact, organizational goals are relatively stable; coalition members develop mutual control-systems, such as budgets or allocations of functions and then bargain within these systems. As

they pointed out, when "precedents are formalized in the shape of an official standard operating procedure or are less formally stored, they remove from conscious consideration many agreements, decisions and commitments that might well be subject to renegotiation in an organization without a memory" (p. 33).

In the example cases, the task is to implement engineering changes to fix existing problems without introducing new problems. I am not claiming that the actors have only this one particular goal; clearly actors have many goals, both organizational and individual. However, I am analyzing the organization's performance with respect to only certain of these goals.

Coordination depends on the level of analysis. The amount of coordination an analyst observes depends on the level of aggregation of the system studied. Consider again the example of a car manufacturing company. Most of the tasks performed by the materials management group, such as ordering parts, planning the shipment of parts or checking supplies, are coordination methods. If you consider just the materials management group, however, then creating a parts shipping schedule is the goal of the group and is a production function. Therefore, in some ways, many actions are and are not coordination simultaneously.

It is also possible to analyze the same physical actions in different ways for different purposes. For instance, we might sometimes regard each person in a work group as a separate actor, while at other times we might regard the whole group as a single actor in a larger organization. Sometimes we might even choose to regard different parts of the brain of a single person as separate actors (e.g., Minsky, 1987). Similarly, observers may choose to define actions at different levels of abstraction depending on their purposes in analyzing the situation.

However, viewing an organization at some level as a collection of black boxes focuses attention on the tasks necessary to coordinate between those boxes. This is the approach I will take in this thesis. I assume a particular level of division of work and analyze coordination at that level.

This approach is necessary in order to make the problem manageable; the assumption is that if we looked inside the black boxes, we would see more-or-less the same set of issues.

Coordination may be necessary even when there is only a single actor. Clearly, many important coordination situations involve multiple actors, and most previous work (e.g., Malone, 1988) has defined coordination as something that occurs only when multiple actors are involved. Since then, however, I have become convinced that the essential elements of coordination listed above arise whenever multiple, interdependent actions are performed to achieve goals, even if only one actor performs all of them.

1.2 Elements of coordinated situations

To provide a more precise definition of coordination to guide my study, I want to identify the important elements of coordinated situations. (Much of the following discussion is drawn from Malone and Crowston (1990).) I start by considering the implications of the following dictionary definition of coordination (American Heritage Dictionary, 1981):

the act of working together harmoniously.

First, what does the word "working" imply? The dictionary defines "work" as "physical or mental effort or activity directed toward the production or accomplishment of something" (American Heritage Dictionary, 1981). Thus there must be one or more *actors,* performing some *actions* which are directed towards some ends. In what follows, we will refer to the ends towards which tasks are directed as *goals.* Actions may produce something and, even though this definition of work doesn't explicitly refer to them, may require tools or other inputs; we will refer to these outputs and inputs collectively as *objects.*

For example, an automobile manufacturing company might be thought of as having goals of producing several different lines of automobiles and a set of actors, human workers, who use objects, such as machines, raw

material and their own efforts to perform tasks necessary to achieve the goals. As another example, a computer network can be thought of as having goals of performing various computations and a set of actors, computer processors of various types, that use and create data to perform tasks to achieve the goals. In the three examples on pages 20–21, the goal is assumed to be fixing reported bugs. The actors include customers, the response center, different intermediaries and software engineers. The objects are not explicitly mentioned, but include the actors' efforts, various computer systems, programming aids and documentation.

These elements—actors, actions, goals and objects—are not independent. By using the word "harmoniously", the definition of coordination implies that the actions must be performed in a way that results in "pleasing" and avoids "displeasing" outcomes. I refer to these relationships between actions as *interdependencies*. Interdependencies constrain how actions can be performed, causing what I call *coordination problems*. In the three examples on pages 20–21, the problems are 1) avoiding duplicate problem reports, 2) finding the correct engineer to work on a given problem and 3) managing interdependencies between modules of the operating system.

Faced with coordination problems, actors must, in general, perform additional actions to achieve their goals. I call these additional actions *coordination methods*. In the examples, the methods used are 1) looking up the answer in a database rather than resolving the problem, 2) task assignment procedures and 3) consulting with other engineers and getting them to make changes to their modules.

1.3.1 About coordination methods

Before going on, a few observations about coordination methods are in order.

Coordination methods do not always arise in the same order. For instance, if a group finds it very hard to handle the interdependencies that

arise from a particular task decomposition, they may choose to redefine the subtasks in a way that leads to fewer (or easier to handle) interdependencies. (Choosing which coordination methods to apply is itself a coordination problem, or perhaps meta-problem, one that I have not addressed in detail.)

Coordination methods are performed over different time scales. Depending on the time period during which we observe a system we will see different kinds of coordination methods. For example, designing or redesigning an organization (e.g., by selecting actors with particular skills or developing roles for them) is a kind of coordination task that is usually carried out only infrequently; picking a particular actor in a given organization to perform some action may happen quite frequently.

Coordination methods are not always equally important. For instance, in scheduling tasks on a network of computer processors, the resources (i.e., computer processing time) may be already available and the primary issues may involve *allocating* rather than *acquiring* these resources.

Coordination methods may be performed by many actors. In some cases, one actor may perform many coordination methods and present a plan to the other actors who simply follow it. Alternately, each actor may do part of the coordination work itself. The argument is simply that these tasks must be performed somehow in order for the organization to achieve the organizational goals at all.

Coordination methods may themselves require coordination. The above analysis of coordination methods can be performed recursively. An organization needs coordination methods to coordinate the production tasks. It may then need other coordination methods to coordinate these coordination methods. For example, the organization may need to assign particular tasks to actors to be performed; these task assignment tasks may themselves be dependent on some other tasks, requiring additional coordination work to manage those dependencies.

2 Why a cookbook?

In this section I will suggest why I believe a cookbook is the appropriate metaphor for my dissertation.

First, like a cookbook author, I have collected a number of recipes, in my case, recipes for multi-actor action. Lochbaum, Grosz and Sidner (1990) cite Pollack (forthcoming) who defines a recipe as what actors know when they know how to do something; therefore, coordination recipes are what actors know when they know how to coordinate. I believe there are domain independent coordination recipes, similar in spirit to the weak problem solving methods described by Newell and Simon (1972); my cookbook will be a collection of some of these recipes. Of course, the analogy to a recipe should not be taken literally or pushed too far; nevertheless, I feel it is the appropriate metaphor for my dissertation.

In most cookbooks, the recipes are written in a way that makes it clear what ingredients are needed and provides a consistent description of the required operations. Similarly, I have developed a language for writing coordination recipes, drawing on techniques developed for modelling actors in distributed artificial intelligence systems.

Most cookbooks organize their recipes into categories such as entrees, appetizers or desserts. The categories of coordination methods I describe include, for example, choosing tasks to achieve organizational goals, choosing actors to perform the tasks and managing different kinds of dependencies between tasks. I believe that other coordination recipes can easily be added to these categories. As with a cookbook, future researchers may not agree on these exact categories, but even if the precise divisions change, the broad outlines are likely to remain the same.

At a more detailed level, certain recipes are composed of simpler components. For example, a pie includes a pie crust and a filling; many entrees are composed of some kind of meat and a sauce. I have similarly decomposed some coordination methods; for example, choosing an actor to

perform a task involves identifying which actors could perform the task, selecting a particular actor and getting that actor to do the task.

Within each category of components, a cookbook may suggest a number of alternatives, providing, for example, a selection of meats, sauces, pie crusts or fillings. The cookbook may help by suggesting which choices fit together well (e.g., a cherry pie and a flaky pastry crust) or identifying exceptions to the rules (e.g., an entree that needs no sauce). Similarly, in the organizations I studied I have identified, for example, several ways to identify actors (e.g., by knowing a particular actor or by asking someone else, including the actors themselves) and to choose between them.

Many cookbooks suggest how particular dishes fit together in a consistent menu. Though beyond the scope of this thesis, I am investigating what I call coordination strategies, that is, consistent patterns of coordination methods. For example, one approach to coordinating a group is to identify all necessary tasks and dependencies and create a plan addressing them that tells each actor what to do, thus eliminating the need for the individuals to coordinate themselves. An alternative approach is to ensure that each actor has the necessary knowledge to determine for itself the correct actions it should take.

Finally, like any cookbook, my collection of recipes is incomplete. My cookbook includes the coordination mechanisms I observed in the organizations in my study, but the individuals I studied certainly did many things other than coordinate and other kinds of organizations may be coordinated in ways I have not yet observed. In particular, I focused on the coordination of on-going processes in established work organizations; it seems likely that additional recipes will be necessary to describe individuals working in other kinds of environments.

2.1 Uses of a cookbook

Even with its limitations, a cookbook is useful. Similarly, I believe that the recipes in my cookbook will be useful as a basis for analyzing an existing

organization or suggesting alternative organizational structures. For example, in my framework, choosing a particular engineer to work on a change to a part and a supplier to build the new part are both examples of the same coordination task (choosing an actor to perform a task) even though they are typically handled quite differently. This analysis suggests at least examining the implications of treating engineers as if they were suppliers and vice versa.

Knowing about different recipes is particularly useful when the raw materials and processes available suddenly change. We are going through such a change now, as computation and communication become orders of magnitude cheaper. For example, having a collection of the recipes people customarily eat would be useful if we try to imagine how these foods would change if microwave ovens suddenly became widely available. At first, these new tools might be used only for simple adaptations of existing recipes, providing a faster way to make such traditional dishes as baked potatoes, popcorn or oatmeal. As people developed more experience with the new tools, however, one would expect to see entirely new kinds of dishes that take advantage of the special characteristics of microwave ovens.

I expect to see similar changes in use of coordination recipes due to the increased availability of information technology. At first, people will do more or less the same things, except with the technology. For example, the initial uses of an electronic-mail system will likely be simply to replace paper memos or phone messages, without fundamentally altering the way the organization works. Eventually, however, new forms will evolve that depend critically on the computer. To some extent, of course, this has already happened. Most banks, for example, would be unable to conduct business without their computer systems. However, as personal computers become more widely used, I expect the same to happen to coordination processes beyond routine transaction processing.

3 Overview of the thesis

At this point, I will briefly describe the rest of the dissertation and outline the thesis of this thesis.

3.1 Modelling coordination

In Chapter 2, I review the artificial intelligence literature about representing actors and actions. Based on these ideas, I present a technique for modelling the coordination methods of a group performing a complex task. I focus in particular on *coordination knowledge*—knowing how to coordinate—as opposed to *production knowledge*—knowing how to do a particular job, in other words, the additional knowledge an individual working in a group needs to know to be an effective member of the group beyond simply knowing how to do his or her individual job.

3.2 Case studies

To test, informally, the utility of the modelling technique and to develop a typology of coordination needs, I performed a field study of the coordination processes of three organizations. In Chapter 3, I discuss the design of the field study I performed and describe how I analyzed the data I collected.

March and Simon (1958) note in the postscript to their book (p. 212) the problem of identifying the program that an organizational unit uses from observation of the behaviour of members of the unit. My approach is to consider the communications between actors as actions taken by the actors to carry out a plan and achieve some goal. Therefore, I first record the communication between actors and the kind of information processing each actor performs. I then develop models of each actor in terms of its knowledge about the task domain, the organization and the other actors. These individual models can then be tested together to ensure that the simulated

behaviours are consistent with the communications patterns actually observed.

I chose to study the engineering change processes of three organizations: an automobile company, a commercial jet aircraft company and the system software development organization of a computer company. I chose engineering change processing for two reasons. First, it is a very coordination-intensive process that requires individuals in engineering and manufacturing to work together to be effective. Second, change management is done in some form by all manufacturing companies. The three companies were chosen partially on the basis of access, but mostly because they seemed to span a sufficient variety of manufacturing processes to offer interesting comparisons with enough overlap to provide some replication.

Chapters 4, 5 and 6 present the results of the study in three sites. These cases also provide the only detailed descriptions of the engineering change process I know of in the engineering management literature.

3.3 Typology of coordination methods

In Chapter 7, I develop a typology of coordination methods organized by the coordination needs they meet. This typology describes the world in terms of tasks and objects—including actors—needed to perform and affected by the tasks. Coordination methods are necessary to manage the relationships between these items, for example, resolving conflicts between different goals or assigning resources to particular tasks.

For each element of the typology, I present a variety of coordination methods and the specific coordination recipes necessary to perform those tasks. These recipes have been drawn from three sources: from the cases studies, from considerations of possible distributions of the elements of the coordination recipes and from the literature.

I conclude in Chapter 8 by summarizing my findings and discussing some possible future research directions.

3.4 Contributions

I believe this dissertation makes three contributions. First, I develop a technique for representing the knowledge, capabilities and goals needed by individual actors to perform coordination methods. A key feature of this technique is that it allows an analyst to reason about ways these items can be distributed among actors. Second, I develop a typology of coordination problems based on the kinds of interdependencies that must be managed in a system with multiple goals, actions and actors. Finally, for each kind of coordination problem, I have found a variety of coordination methods, each expressed as a set of coordination recipes.

4 Related work

In this section, I will briefly present relevant work from organizational studies and artificial intelligence and discuss limitations of each body of work for my study.

The study of coordination lies at the intersection of numerous fields. One characteristic that distinguishes these different research streams is the nature of the systems studied. Researchers in organizational studies have investigated the coordination of systems of human beings, from small groups to large formal organizations (e.g., Galbraith, 1977; Thompson, 1967). Economists have studied coordination in markets composed of independent profit-maximizing firms (e.g., Debreu, 1959). A branch of economics, agency theory, suggests how costs arise from the division of ownership and control at many levels of analysis (e.g., Holmstrom, 1979; Jensen and Meckling, 1979; Ross, 1973). Control theory provides principles for coordinating mechanical systems made up of separate components; system dynamics applies these principles to the analysis of many different kinds of natural and man-made systems. Computer science, and in particular DAI, has recently begun to investigate ways to coordinate systems made up of interacting processes (e.g., Fox, 1981; Hewitt, 1986; Huberman, 1988; Miller and Drexler, 1988; Smith and Davis, 1981).

Other disciplines suggest ways that systems can appear to be coordinated, even though the individual actors act independently and in a non-purposeful fashion. For example, evolutionary biology and ecology show how the forces of natural selection can result in seemingly coordinated systems of animal behaviour (Franks, 1989; Seeley, 1989). Neural networks exhibit organized activity from the sometimes random actions of very simple subunits. (I describe these situations as seemingly coordinated because by my definition of coordination, actions can only be said to be coordinated if they are taken to achieve some goal; in these cases, it seems forced—although not impossible—to view the actions as goal-directed.)

I believe that a more integrated view of coordination is necessary because these various literatures focus narrowly and on different issues. Agency theory, for example, has as a central concern problems that arise when more than one person works on a task. The primary focus, however, is how to solve problems caused by possibly conflicting goals between agents and principals; it does not address the many problems that exist even when there is no goal conflict. Past research in organizational studies has focused mostly on issues of decision making and has not been very specific about the source of dependencies between tasks or how different coordination mechanisms work. Research in artificial intelligence has been very formal, but not necessarily related to human organizations. In its attempts to find generalizations that apply across disciplines and across levels of analysis, coordination theory resembles earlier work on systems theory and cybernetics (e.g., Beer, 1967; Boulding, 1956; Emery, 1969; von Bertalanffy, 1950; Wiener, 1961).

Given the large number of potentially relevant bodies of literature, one key issue I face is where to anchor my research. I have chosen to focus on two sets of theories. I have drawn first on organizational studies, in particular, on the Carnegie school and its focus on the organization as a decision maker. The second body of literature is artificial intelligence, since it provides useful formalizations of communication and decision making. Also, artificial intelligence's view of the actor as a boundedly rational information processor

is compatible with the assumptions of the Carnegie school. I use the strengths of these two fields to investigate more precisely how people coordinate their actions when they work together.

4.1 Coordination in organizational behaviour

The need for coordination has been explored by many organizational theorists. The usual conception is of a fit between task requirements and organizational structure, where organizational structure is determined in part by the coordination requirements.

4.1.1 Coordination as task design and assignment

March and Simon (1958) review early efforts in a field they call "administrative management theory". In this field the problem is seen as designing an organization to optimally perform a given set of tasks. One formulation of this problem is as an assignment problem: given n people, n jobs and costs c_{ij} for person i to do job j, what is the cheapest way to assign the jobs to the people? A more interesting modification of this problem is to assume a set of activities, $\{S_i\}$, where the cost to do activity S_i is $C(S_i)$ and the cost to do a combination of activities, $S_1 + S_2$, is different than the cost to do them separately (i.e., $C(S_1 + S_2) \neq C(S_1) + C(S_2)$). In this case, the problem becomes one of designing tasks (sets of activities) to be done by individuals to minimize the total cost necessary to do all the tasks.

Complications arise from the fact that the times are not simply additive; that is, if there is a fixed setup cost, it is often cheaper for one person to make two of something than for two people to make one each. Organizations also place constraints on the kinds of tasks that can be designed. For example, people in a certain department may work only on specific activities, such as a particular product (specialization by purpose) or know only particular processes, such as engineering or manufacturing (specialization by process). The most efficient partition that can be designed within these constraints may not be the most efficient of all possible partitions (March and Simon, 1958, p. 23-24).

March and Simon (1958) note a key problem with this formulation of coordination:

> One peculiar characteristic of the assignment problem, and of all the formalizations of the departmentalization problem in classical organization theory, is that, if taken literally, problems of coordination are eliminated. Since the whole set of activities to be performed is specified in advance, once these are allocated to organizational units and individuals the organization problem posed by these formal theories is solved (p. 25-26).

In other words, coordination is more than simply dividing up the work and assigning it to actors.

4.1.2 Coordination as choice of actions

March and Simon (1958) go on to offer their own view of coordination, in which communication between actors may be necessary for several reasons:

> (a) the times of occurrence of activities may be conditional on events external to the organization or events internal to the organization;
> (b) the appropriateness of a particular activity may be conditional on what other activities are being performed in various parts of the organization;
> (c) an activity elaborated in response to one particular function or goal may have consequences for other functions or goals (p. 27).

In this case, "the problem of arranging the signalling system for interdependent conditional activities is the coordination problem" (p. 28). March and Simon identified a number of coordination mechanisms that address this problem.The simplest mechanism is self-containment of task: one way to simplify a problem is to break it into nearly independent parts so each subunit can solve one problem without paying attention to what the others do (p. 151). This compartmentalization has costs, however. March and Simon (1958) cite research by Marschak and Radner (1954) and Marschak (1955) that concludes that forms of departmentalization that are advantageous in terms of self-containment are costly in terms of skill specialization (and vice versa).

The division of work by process leads to greater interdependence. Ways of reducing the interdependence they identified for manufacturing include using semi-manufactured goods, interchangeable parts and buffer inventories. Most manufacturing operations start by reducing highly variable raw materials to a more homogeneous semi-manufactured good, such as iron ore to pig iron or cotton to thread. This reduction greatly simplifies the design of subsequent processes that can assume a homogeneous input. Interchangeable parts reduce the need for groups to coordinate about the goods they use, since the fit is assured. Finally, buffer inventories reduce the need for groups to coordinate the timing of their processes (p. 160).

Ways to increase coordination include schedules and feedback when things change. Schedules simply spell out the interactions expected between different groups. When events arise that can not be anticipated and planned for, groups must communicate changes to plans (p. 160).

As the organization grows, the marginal advantages of process organization become smaller and the coordination cost grows, resulting in a shift from process to purpose organizations (p. 29).

4.1.3 Coordination as a response to topologies of interdependence

Thompson (1967) builds on the "Simon-March-Cyert" tradition. He identifies three kinds of interdependence: "pooled" (each actor contributes to and depends on the whole); "sequential" (one actor depends on another); and "reciprocal" (the outputs of one actor are the inputs of the other and vice versa). These kinds of interdependence are increasingly hard to coordinate because they contain increasing degrees of contingency. With pooled interdependences, the actors can act separately; with sequential, each must readjust if the previous one does not do what was expected; and with reciprocal, all must readjust whenever one changes (p. 55).

Thompson identifies three kinds of coordination, drawn from March and Simon: "standardization", "plan", and "mutual adjustment" (p. 56). He then equates the requirements of his three kinds of interdependence with

these different kinds of coordination. Actors who have pooled interdependences coordinate by standardizing their actions, using rules to constrain their actions to be consistent with others. Actors who have sequential interdependences use plans to establish schedules for their actions. Actors who have reciprocal interdependences communicate new information during their actions, allowing each to adjust to the other.

Thompson lists the four usual ways of making departments—by purpose, process, clientele or geographic area—and points out that it is impossible to form groups that meet all of these constraints at once. He notes that increased coordination requires increased communication and decision making, making these mechanisms increasingly expensive. He therefore argues that the primary coordination mechanism organizations use is to form departments that group actors to reduce coordination costs. For example, actors with reciprocally dependent positions should be placed in common, more-or-less self-contained groups. Second order groups can then be created, forming a hierarchy, to address interdependences left by the first order groups, and so on (p. 60).

Thompson suggests a number of additional structures to provide further coordination. Groups with pooled interdependences that are not in the same group use standardization, and add liaison positions between the standardizers and the groups. Groups with sequential interdependences not covered by departmentalization use committees to accomplish the remaining coordination. Groups with reciprocal interdependences not covered use task-forces or project groupings (p. 61). As an example of this approach, Thompson analyzes the structure of a bomb wing in terms of the interdependences between different groups (pp. 61–65).

4.1.4 Coordination strategy depends on organizational type

Mintzberg (1979) takes the view that there are certain configurations of organizational variables that are most likely and thus appear as a fit between the organization and the environment. One such variable is the kind of coordination used. He identifies five kinds of coordination: 1) mutual

adjustment, 2) direct supervision and standardization 3) of work processes, 4) of outputs and 5) of skills. Mintzberg claims that each mechanism is most often found in a particular kind of organization and environment. In particular, simple structure organizations use direct supervision, the machine bureaucracy uses standardization of work processes, the professional bureaucracy uses standardization of skills, the divisional form uses standardization of outputs and the adhocracy uses mutual adjustment.

4.1.5 Coordination strategy depends on level of interdependence

McCann and Galbraith (McCann and Galbraith, 1981) suggest that coordination strategies vary along three dimensions—formality, cooperativeness and localization—and that as dependency increases, the amount of coordination necessary increases and as conflict increases, coordination strategies chosen become increasingly formal, controlling and centralized. They therefore propose a two-by-two matrix showing conditions under which organizations will choose to coordinate by rules, mutual adjustment, hierarchy or a matrix structure, but they do not describe the coordination processes themselves in any more detail.

4.1.6 Coordination as patterns of information processing

Early authors are often rather vague about what the coordination mechanisms they identify do. The information processing (IP) school suggests that these mechanisms are useful because they increase the organization's information processing capacity. Tushman and Nadler (1978, p. 292) outline three basic assumptions of IP theories:

(a) organizations must deal with work-related uncertainty;
(b) organizations can fruitfully be seen as information processing systems; and
(c) organizations are composed of individual actors.

In this view, organizational structure is the pattern and content of the information flowing between the actors and the way they process this information. Tushman and Nadler (1978) go on to hypothesize the need for a

fit between the organization's environment and its information processing capacity.

Galbraith (1974; 1977) expands on their work, explicitly considering an organization's need to process information to reduce environmental uncertainty and strategies by which it could achieve this goal. He suggests four strategies: 1) increased slack and 2) creation of self-contained tasks, as ways to decrease the need for information processing and 3) creation of lateral ties and 4) vertical information systems as ways to increase the organization's capacity.

4.2 Coordination in artificial intelligence

The design of distributed computer systems raises a number of interesting coordination issues. The following three example systems illustrate a range of approaches to these problems.

4.2.1 Communication through a shared blackboard

The Hearsay-II speech understanding system (Erman, et al., 1980) uses a blackboard model for coordinating independent actors, each with specialized knowledge about a problem (in this case, speech recognition). The actors communicate through the blackboard, a common data structure on which actors post results or preliminary hypotheses which can be used as input by other actors. For example, actors processing low-level sound data can post hypotheses about the phoneme being uttered, which can be used by other actors to make guesses about the word. Other actors can then check that the hypothesized word makes sense or suggest alternative hypotheses which should be tested. The blackboard structure allows great flexibility in the construction of modular systems combining diverse sources of knowledge (Barr and Feigenbaum, 1981, p. 343).

4.2.2 Task assignment through contracts

Contract nets (Davis and Smith, 1983) are a mechanism for the cooperative solution of problems by a decentralized and loosely coupled

collection of problem solvers. Davis and Smith focus on the issues of partitioning and decomposition, rather than assuming a partitioned task; in particular, they do not assume that actors start with a map of the subproblems and their interdependencies. Complications arise from the fact that actors have only a local view of the problem but global coordination is necessary.

To solve this problem, Davis and Smith use the metaphor of a group of human experts working together on a novel problem and design a problem solving protocol that the actors follow. Each actor independently works on some part of the problem. If the actor needs more resources (for example, special hardware for a particular kind of problem or just additional processor time) it can put the task up for bids by broadcasting a description of the task. Free actors can bid on the task, indicating, for example, how long they estimate they would take to complete it. The first actor chooses an actor based on the bids received and assigns the task (or perhaps decides to continue working on the task itself). Assignment is thus based on an interaction between caller and respondent, rather than the caller or the respondent simply choosing the next actor or task.

4.2.3 Partial global plans

Partial global plans (Durfee and Lesser, 1987) are a way for independent actors to work together on common goals. Each actor makes plans at a high level of abstraction, showing what it intends to do, and exchanges these plans with other actors. Based on these plans, actors can identify goals they share with others and build partial global plans to achieve those shared goals. For example, if two actors are working on overlapping problems, they can decide when to share partial results or to reorder the tasks they perform.

4.3 Conclusion

Earlier authors in organizational behaviour note such key factors as the limited rationality of human beings, which lead them to consider explicitly the way people and organizations gather and process information. Their analysis, however, emphasizes factors such as the steps involved in decision-

making and does not focus much at all on the amount and kinds of communication between different actors. As a result, they are generally vague about why coordination is necessary and how coordination is actually achieved.

In general, the authors I have considered do not define coordination. March and Simon (1958) are most explicit; most simply give example of problems requiring coordination and a list of mechanisms.

The assumption that organizations choose an optimal strategy is a general problem with models that hypothesize a fit between structure and the environment. If one assumes only that organizations choose a satisfactory system, then prediction becomes much more difficult (March and Simon, 1958, p. 145-146), particularly because it is unclear what it would mean for a system to be satisfactory. Gresov (1988) notes that misfits may also occur because the environment may change after the organization is designed and organizations have trouble shifting quickly and completely to meet new environments.

Finally, these authors are not specific about how the mechanisms they describe provide coordination. Thompson (1967), for example, mentions committees as a means of coordinating groups, but he does not say what the committees do, who should be on them, what they talk about—in short, exactly what committees do to coordinate. While the mechanisms suggested are certainly useful and used, it is difficult to gauge the extent of their applicability. In particular, it is unclear how computer systems can be used to help provide coordination or what mechanisms will be useful in coordination-intensive organizations.

The research in AI is in many ways in an opposite state. Definitions of coordination and interdependence and descriptions of coordination mechanisms are very formal, because the algorithms must be sufficiently precise to be computerized. However, the problems attempted typically have fairly simple interactions and operate in a well-understood environment. Furthermore, researchers can design systems that concentrate on solving only

the actual problem, taking the creation of the organization and of understanding the problem largely for granted.

My approach to studying coordination merges the strengths and avoids the weaknesses of these two disciplines. Focusing on coordination problems and coordination methods provides a more precise definition of coordination. Models of the information-processing behaviour of individuals in organizations, as discussed in Chapter 2, provide a more specific way to describe these mechanisms. Studying real organizations, as discussed in Chapter 3, provides a rich source of data to ground theory development.

2

MODELLING COORDINATION

What AI has to contribute to psychology is exactly this experience with modelling processes.

—Schank and Abelson, *Scripts, Plans, Goals and Understanding*

Recipe language is always a sort of shorthand in which a lot of information is packed, and you will have to read carefully if you are not to miss small but important points.

—Child, Bertholle and Beck, *Mastering the Art of French Cooking*

"Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

—Lewis Carroll, *Through the Looking Glass*

Developing techniques to study how members of groups coordinate their actions is the first contribution of my dissertation. My approach is to observe organizations in action and develop models of the actors in the organization in a knowledge-based formalism (in principle, in enough detail to be executable on a computer).

1 Why model?

One method other information-processing-based disciplines use to gain insight into complex behaviours is to imagine how a computer could be programmed to reproduce them. Computer models of organizations can provide at least three benefits for the study of coordination in organizations.

First, models provide formal representations of assumptions about the process being modelled. In cognitive psychology, for example, computer models

of learning or memory embody theories about human information processing and can be used to generate further empirically testable hypotheses. Cyert and March (1963) take this approach to the study of organizations. In their analysis of the processes firms use to make pricing decisions, the "process is specified by drawing a flow diagram and executing a computer program that simulates the process in some detail" (p. 2). In this sense, a model of the process is a statement of a hypothesis about how the organization behaves.

Cyert and March point out an important difference between models built in the physical and social sciences. In the physical sciences, the properties of the subunits are usually known to some high degree and the modeller is interested in the properties of the whole system. For example, a meteorologist might attempt to predict the weather (a system property) using a model that describes the properties of the oceans, the atmosphere, etc. In social sciences, however, the researcher can usually observe the whole system and is instead interested in hypothesizing about the behaviour of the subunits and the relations between them. The procedure is to construct a model that specifies the behaviour of the subunits and check that the model exhibits the same behaviour as the system (p. 317).

Second, models can be used to abstract from and simplify complex systems such as the organizations I studied. As Yourdon notes, "we can construct models in such a way as to highlight, or emphasize, certain critical features of a system, while simultaneously de-emphasizing other aspects of the system" (1989, p.65). Artificial intelligence research and in particular the developing field of distributed artificial intelligence (DAI, e.g., Bond and Gasser, 1988b; Gasser and Huhns, 1989; Huhns, 1987; Huhns and Gasser, 1989) can contribute interesting formalisms for understanding and representing the actions of human organizations.

Finally, computer systems provide a much more tractable method than field studies for investigating certain questions about organizations. For example, it is possible to perform true experiments comparing systems of coordination using computer models (e.g., Durfee, 1988). In a sense, DAI is (or can be) an experimental branch of organizational science.

2 What to model

All models include only selected aspects of the observed phenomena and omit many other possibly important elements. This simplification is necessary: if the models are not simpler than the system they represent, then there is little to be gained by studying them instead of studying the system itself. The choice of factors to include or to omit depends on which behaviours of the system you wish to model and on the assumptions you make about the system.

The choice of behaviours to model obviously depends on the questions you hope to answer with the model. The choice of a set of assumptions is a more difficult problem. The truth of an individual assumption is in some ways irrelevant; some simplifying assumptions may be very useful despite being in point of fact inaccurate. (If, however, the models are fundamentally different from the system, then in some ways it is only a coincidence that the behaviour of the model and system are the same.) The best test of a set of assumptions is whether models built using those assumptions exhibit behaviours similar to the system in ways that provide useful insights.

The development of physics provides a good example of these issues. Philosophers such as Aristotle concentrated on the obvious physical fact that things in the real world stop moving unless you keep pushing them. As a result, they made only limited progress in understanding the world. Newton succeeded in progressing much further by hypothesizing an obviously impossible frictionless world as a basis for his laws of motion and adding friction as a secondary force.

I hope to make some progress by similarly ignoring true but difficult-to-model facts about the world and concentrating on some parts I do understand. This done, some complex omitted issues may clarified by what I learn.

The key issue in modelling is the choice of which elements of the observed phenomena to include and which to omit. Which variables should appear in the model? How should they appear? What are the appropriate values? Different modellers have chosen different answers to these questions. To suggest which

features are important to include and which are unimportant details, it is necessary to have a strong theoretical view of the organization.

In the remainder of this section I will present the assumptions about organizations, individuals and the organizational context that underlie my efforts to model coordination processes.

2.1 Information processing view of organizations

As discussed above, coordination seems to be primarily an information-processing task. For my study I therefore adopted the information processing (IP) view of organizations (e.g., Galbraith, 1977; March and Simon, 1958; Tushman and Nadler, 1978) because of its focus on how organizations process information. In this view, organizational structure is the pattern and content of the information flowing between the actors and the way actors process this information. For example, McKenney, Doherty and Sviokla (1986) performed such an analysis in a software firm, tracing the flow of information and drawing flow charts to describe the processing involved in certain tasks.

The major problem with this approach is that the concepts discussed are still very aggregate. Earlier researchers viewed information almost like a fluid, and uncertainty, its lack. An organization's structure then is like plumbing that directs the flow of information to where it is needed to reduce uncertainty. Such general factors are, as Galbraith noted, very difficult to measure. Such simplifications are useful for general studies, but permit only general conclusions. A more detailed analysis would attempt to characterize the content of messages that comprise the flow of information and examine the processing that these messages require.

I make such an analysis. Instead of looking simply for the presence of information or uncertainty, I attempt to identify the content and purpose of the messages being exchanged and the actions that these messages trigger in the agents. I therefore examine the sources and users of data, the types of messages sent and received, and the actions actors take when they receive certain messages.

Like other information processing theorists, I treat organizations as a collection of intercommunicating *actors*, where in principle the actors can be human beings, computer systems or any other kind of information processor. (In the organizations I studied, the actors are most often human beings.) The individual actors who make up the organizations are assumed to be intendedly rational problem solvers who communicate (and take other actions) in order to achieve their individual and organizational goals. My assumptions about the individual actors are largely those of information processing psychology (Newell, 1979; Newell and Simon, 1972).

However, simply modelling the communication and information-processing alone would be insufficient. Such a model could not explain why the actors communicate as they they do and not in other equally plausible ways. As Newell and Simon (1979) noted in their analysis of individual problem solving, the flowcharts they made of individuals' processes just appeared, with no real explanation as to their origins.

Also, since such a model captures only on-going communications, they can not fully analyze organizations where actions have become routinized. In these cases, much of the interesting communication may have taken place in the unobserved past with coordination at time of observation dependent on the actors' shared agreements.

To make the coordination mechanisms more visible, my analysis must overcome these limitations, as Newell and Simon (1972) did with their abstraction of problem spaces. I therefore seek to model the unobservable goals that (presumably) lead to the observed communications.

2.2 Simplifying assumptions

In general, coordination of a group of actors may be complicated by the fact that actors are not perfectly rational (indeed, they may frequently act in apparently irrational ways). Actors without perfect memories, for example, may need to do additional work to ensure that they actually perform the actions they

plan to. Groups of actors may not share the same goals or even a common language.

However, studying routine tasks performed by individuals in work organizations allows me to make several key simplifying assumptions.

Actors have defined roles. I model actors in the organization in terms of the actions they take in response to a small set of different kinds of messages and assume that their capabilities are known to the other actors. March and Simon argue similarly, noting that to simplify the world, organizations use a set of standard responses, a classification of cues and rules to map cues onto responses (1958, p. 164). Furthermore, organizations often come with roles that define kinds of messages understood by the person in the role, ways of processing messages and expectations about tasks to be done.

Furthermore, a good deal is known about the special abilities and characteristics of members of the organization. As March and Simon point out, roles in formal organizations tend to be, "highly elaborated, relatively stable, and defined to a considerable extent in explicit and even written terms. Not only is the role defined for the individual who occupies it, but it is known in considerable detail to others in the organization who have occasion to deal with him" (p. 4).

Actors have a shared language and use defined communication channels. I assume that communication between actors is mostly reliable. In particular, I do not worry about the need for actors to translate messages, but rather assume that they can quickly recognize that a message is of a particular type. For routine tasks, communication between actors seems to be mostly reliable. Since I am focusing on coordination rather than communication problems, I choose not to emphasize these issues and to handle them simply where they do arise.

For the kind of organizations I studied, this focus seems reasonable. Actors in formal organizations use a common language for task-related communication. As March and Simon note, formal organizations differ from other groups in that they are more specific with respect to channels of

communication and content of communications. Communication between members of the organization is by means of special and precise common technical languages (1958, p. 164).

Actors are cooperative. Finally, I assume that actors are essentially co-operative, that is, that their goals mostly do not conflict and when asked to do something, they do it. For the kinds of routine tasks I studied, this assumption seems reasonable. Task-related goals are usually not questioned; instead they are accepted as part of the actor's role in the organization and requests consistent with these roles are accepted. Organizational goals are internalized, to various degrees, by the individual actors.

3 My approach to modelling coordination processes

A large number of techniques have been developed to represent processes. For example, Marca and McGowan (1988) present a technique called SADT™ (for Structured Analysis and Design Technique) which represents processes and the things that flow between them by boxes connected by arrows. Standard textbooks, such as Yourdon (1989) describe many other techniques, including flow charts and data flow diagrams.

These techniques are certainly useful for describing processes and, as I describe in the next chapter, I use a similar technique in the initial stages of my data analysis. However, because these techniques do not explicitly represent the goals of the process but only the actions taken, they make it difficult to reason about alternative ways to achieve the same goals.

I therefore draw most heavily on research in distributed artificial intelligence (DAI) for techniques to represent the actors and the goals they attempt to achieve. Representing actors and their knowledge is a very active research area in DAI and it seems clear that techniques developed in this area can be applied to modelling human organizations.

I adopt the reductionist view that groups of actors do nothing more than what the individual actors do and focus exclusively on modelling the

information-processing behaviour of the individual actors that comprise the organization (Prietula, et al., 1990). I model each actor as an independent goal-directed problem solver. Each actor is assumed to have its own knowledge about the world; actors can communicate but do not directly share memory. Each actor independently attempts to achieve its goals, given the state of the world as it knows it, by taking actions that affect that state.

Much of the work in representing actors has been directed towards the development of autonomous actors and many of the issues studied are important for implementing simulations of organizations. For the purposes of this thesis, however, I am concerned initially with simply developing a language for describing organizations, so I will mostly gloss over questions of implementing the kind of reasoning I describe. I believe that the models I develop could be implemented with some extra work, but I do not take this step in this thesis.

3.1 A brief introduction to logic

Each actor is modelled by a set of well-formed formulas in first-order predicate calculus, extended as needed. Using logic as a basis for a representation scheme is widespread and although it is not universally accepted, it is sufficient for my purposes and I will not examine the alternatives here. (For a better defence of the utility of logic see Hayes, 1977; McDermott, 1978; Moore, 1982). It should be noted that none of my results depend crucially on the use of logic as a representation.

In this section, I will briefly describe first-order predicate logic. This description is based on Davis (1990, Chapter 2).

Formulas in first-order logic are composed of constant symbols, variables, function symbols, predicate symbols, Boolean operators such as $\wedge$ and $\vee$, and the quantifiers, $\forall$ and $\exists$. Constants, variables and functions are called *terms*; predicates, Boolean combinations of formulas and quantified formulas are called *formulas*. The arguments to a function or predicate must be terms, not formulas.

For example, in the blocks world[1](Winograd, 1972), constants include the table and various blocks A, B and C (`table`, `a`, `b`, `c`); functions include the colour or weight of a block (`colour(Block)`, `weight(Block)`). Predicate symbols, representing relationships between terms, include the fact that one block A is on top of another block B (`on(a, b)`) or that the top of a block is clear (`clear(Block)`). As a second example, the fact that an actor has some object is represented by the two-place predicate `have(actor, object)`. (In these examples, atoms in lower case (such as `table`, `a`, `b`, `c`) are constants; atoms in upper case (such as `Block`) are variables.)

We represent the fact that all blocks are red by the quantified formula:

```
∀Block: block(Block) ⇒ colour(Block) = red;
```

that some block is on the table by:

```
∃Block: on(Block, table).
```

A variable that appears in the context of a quantifier is said to be *bound;* I adopt the convention that unbound variables are considered to be universally quantified.

Sorted logic. In general, functions and relations are only defined on objects of a particular sort. For example, in our blocks world, only blocks have a weight; the table (let us assume) does not. To simplify expressions by eliminating the predicates necessary to assure that all variable are of the correct sort, I will use a *sorted logic* (Davis, 1990, p. 44–45), which is similar to a typed programming language. The arguments to each function and relation and the value of each function will be declared to be of a particular sort. The sorts of quantified variables are assumed to be implicitly declared by the functions and predicates in which they are used (Davis, 1990, p. 45).

---

1 A blocks world is a problem domain used in several early AI systems (e.g., SHRDLU (Winograd, 1972)) consisting of blocks of various shapes and colours on a table, manipulated (usually only conceptually) by a robot arm.

Situations. To allow reasoning about the order of events and temporal constraints, assertions may have different values in different situations. Following Davis (1990, p. 188) I have modelled varying values as *fluents* and varying relations as *Boolean fluents* or *states.*

For example, in the blocks world, `on(a, b)` would be the state of block a being on block b; `colour(a)` would be the fluent of the colour of a block. The predicate `true-in(situation, state)` asserts that the state holds in the given situation; `value-in(situation, fluent)` gives the value of a fluent in the given situation. Situations are ordered; that is, if $situation_1$ occurs before $situation_2$ then

$$situation_1 < situation_2.$$

A situation may be associated with a clock time (`value-in(situation, clock-time)`) to allow reasoning about durations of intervals. We can also talk about intervals of situations and the value of a fluent during an interval. For example,

$$S \in \texttt{interval} \Rightarrow \texttt{red=value-in(S, colour(a))}$$

says that block A was red throughout the given interval. `start(interval)` and `end(interval)` are functions returning the starting and ending situation of the interval, respectively.

In the remainder of this chapter, I discuss the specific elements I have choosen for my representation language. In particular, I discuss the way I represent actors' knowledge, goals, and actions and review the literature that led me to these choices. I draw particularly heavily on work by Morgenstern (1988) and Davis (1990). I conclude by discussing the format in which I present these models in this thesis.

4 Modelling knowledge about the world

Each agent has some set of knowledge, including knowledge about the world in general, about the problem domain and about the organization and

other actors. To say that an actor knows some fact is to express a relation between the actor and the fact known.

I represent what each agent knows about the world using a representation such as `know(actor, fact)`. This representation can distinguish between different states of incomplete knowledge (Cohen and Perrault, 1979), such as $agent_1$'s knowing that:

1) $agent_2$ knows that the train leaves from gate 8;
2) $agent_2$ knows that the train has a departure gate; and
3) $agent_2$ knows what the departure gate is for the train.

In this representation, these statements can be represented as follows:

1) `know(agent`$_1$`, know(agent`$_2$`, leaves-from(train, gate 8)));`
2) `know(agent`$_1$`, know(agent`$_2$`, ∃X: leaves-from(train, X)));` and
3) `know(agent`$_1$`, ∃X: know(agent`$_2$`, leaves-from(train, X))).`

It should be noted that these representations of an actor's knowledge are not legal expressions in first-order predicate calculus, because `fact` known is a formula, not a term. (This restriction applies also to other sentential verbs such as want or believe.) Furthermore, `know` must be *referentially opaque*, that is, what is known depends on the representation of the fact, so equal values can not be substituted within the expression. For example, Oedipus knows he married Jocasta:

```
know(oedipus, married(oedipus, jocasta)),
```

but despite the fact that Jocasta is his mother:

```
jocasta = mother(oedipus),
```

he does not know that he married his mother:

```
~know(oedipus, married(oedipus, mother(oedipus))),
```

(Davis, 1990, p. 54–55).

There are two possible solutions to these problems. First, first-order predicate calculus can be extended with a modal operator, `know`, and axioms for reasoning about it. In this case, `know(agent, fact)` is a valid statement of a modal logic (for details, see Chellas, 1980; Davis, , 1990; Halpern and Moses, 1985)). Alternatively, we can include quotation in the logical language and represent knowledge by predicates of the form `know(actor, 'fact')`, where `'fact'` is a string representing the logical statement `fact`. Using quotation allows for greater flexibility in what can be represented, but is difficult to axiomatize consistently (in particular, it is possible to construct self-referential statements such as "this statement is false").

Most AI researchers use modal logic, although some have argue for quotation (Morgenstern, 1988). Since the difference concerns the details of the semantics of these statements, I will use the former representation, while leaving open the possibility of using the second approach for actually implementing the models.

4.1 Assumptions about knowledge

Given this formalism, we can express what assumptions we will make about the knowledge of the actors. (This discussion of the axioms of knowledge is drawn from Davis (1990, p. 375).)

The closed world assumption is that all true facts are known:

$$\varphi \Rightarrow \mathtt{know(actor,}\ \varphi\mathtt{)}$$

and assertions that are not known are known to be not true:

$$\sim\mathtt{know(actor,}\ \varphi\mathtt{)} \Rightarrow \sim\varphi.$$

These assumptions are particularly unrealistic and will not be used in this thesis.

Actors are almost always assumed to know a basic set of axioms and the transitive closure of their knowledge:

$$\mathtt{know(actor,}\ \varphi\mathtt{)} \wedge \mathtt{know(actor,}\ \varphi \Rightarrow \psi\mathtt{)} \Rightarrow \mathtt{know(actor,}\ \psi\mathtt{)}.$$

In other words, actors know any statement that can be derived from the statements they know, or alternately, actors are perfectly rational. This assumption is also rather unrealistic. However, it turns out to be rather difficult to model restrictions on rationality (Morgenstern, 1988, pp. 107–111) and so this assumption is usually kept.

A common assumption is that all facts known by an actor are true (the principle of veridicality):

$$\texttt{know(actor, } \varphi\texttt{)} \Rightarrow \varphi.$$

This principle distinguishes knowledge from belief; beliefs may turn out to be not true. (More accurately, knowledge is true and justified belief, since having a belief that turns out to be correct by chance, such as betting on a winning horse in a horse race, should not count as knowledge (Davis, 1990, p. 374).)

This principle seems unrealistic, since actors' knowledge is often wrong. To reason in terms of beliefs, however, requires a good model of what it means to believe something and a method for drawing plausible inferences from beliefs. These are both rather difficult problems that cannot be solved in this thesis. Therefore, in my models I will speak in terms of knowledge and monotonic inferences from that knowledge. For the kinds of processes I am studying, this approximation is workable, although a more complete theory would include some system for plausible inference. In the few places where it is important to indicate that some derivation is only a plausible inference, I will use the notation:

$$\texttt{plausible(}\phi, \psi\texttt{)}$$

meaning that given that ϕ is true, it is plausible that ψ is true, without going into the details of how this inference should be implemented (Davis, 1990, p. 101).

Actors are often assumed to know what they know, the principle of positive introspection:

$$\texttt{know(actor, } \varphi\texttt{)} \Rightarrow \texttt{know(actor, know(actor, } \varphi\texttt{)).}$$

Without this assumption, it is difficult for an actor to reason about what information it needs to perform an action. Actors are sometimes assumed to know what they do not know, the principle of negative introspection:

$$\sim\texttt{know}(\texttt{actor}, \varphi) \Rightarrow \texttt{know}(\texttt{actor}, \sim\texttt{know}(\texttt{actor}, \varphi))$$

but this assumption is unrealistic for real world problems and will be dropped.

The problems that arise from this set of assumptions are more serious for automating reasoning about knowledge than for simply representing it. These issues must be resolved before the models can be used as the basis for a computer simulation of an organization, but for the purposes of this thesis, they will generally be overlooked.

4.2 Organizational knowledge

As a basis for communication, actors need models of the organization that let them reason about their interactions with other actors. For example, to know to ask another actor for help, actors must be able to reason about other actors' knowledge and capabilities. In the examples discussed in the introduction, customers know about the capabilities of the response center; the various intermediaries know which engineer can fix particular kinds of bugs; and software engineers know about each other's responsibilities.

Gasser and Rouquette (1988) suggest viewing an organization as a set of settled (and unsettled questions) about the beliefs actors have about each other. In their view, the basic problem is deciding which actors do what and when. This problem can be resolved either through on-going problem solving or on the basis of conventions and routines (i.e., settled questions).

Issues in representing actors in organizations are reviewed by Bond and Gasser (1988a). They suggest that each agent must know the capabilities and responsibilities of other agents; resources available and demands on the resources; the extent of progress towards a solution; how to communicate, including knowledge of available channels, languages and protocols; what

information will be useful for other agents and should be communicated; and the other agents' beliefs, goals, plans and actions.

In order to work together, the individual actors need to know the task the organization is performing, the the particular subtasks they are doing, the relationships between that subtask and others and the capabilities of other actors in the organization. For each task and organization, the precise information required will vary.

For these reasons, I therefore model each actor's goals, capabilities and knowledge about task domain and its own mental models of the capabilities of other actors. Essentially, I attempted to reverse engineer the coordination (and production) knowledge actors use from the messages they send to other people.

5 Modelling actors' goals

A goal is a desired state of the world, stated as a formula in predicate calculus. For example, in a blocks world, a goal might be to have blocks a, b and c in a stack on the table, that is:

$$\text{on}(a, b) \wedge \text{on}(b, c) \wedge \text{on}(c, \text{table}).$$

For a car company, the goal might be to produce a certain number of car objects during the year. In the examples from the cases, customers can be viewed as having the goal of getting a fix for a problem. Goals should be distinguished from constraints, such as the need to produce the cars with the available equipment or personnel.

I assume the actors have known goals they are trying to achieve. (As discussed above, this assumption seems reasonable for the kinds of organizations I wish to model.) To allow the actors to be able to reason about their goals and about ways to accomplish them, the goals must be explicitly represented by a statement such as:

```
want(actor, goal).
```

Furthermore, we will assume:

```
want(actor, goal) ⇒ know(actor, want(actor, goal))
```

that is, actors know what their goals are.

The set of goals held by the individuals in an organization can be viewed as a *goal hierarchy*. At the top of the hierarchy are broad organizational goals shared by all members of the organization, such as making cars or money. These goals are not operational, however, that is, they do not directly translate into actions that an individual actor can carry out.

Instead goals must be decomposed into subgoals that when achieved achieve the goal. For example, making cars may be divided into steps such as designing cars and manufacturing them; making cars itself is probably only a means to achieve the higher goal of making money. These subgoals are themselves realized by progressively more specific subgoals. At the bottom of the hierarchy are goals which can be directly achieved by individual actors, such as preparing a drawing of or manufacturing a particular part. These achievable goals are usually referred to as *actions*; my representation of actions is discussed in the next section.

Exactly which goals are considered actions depends on the purpose of the analysis: for example, organizational designs are usually stated in terms of the actions of work groups while time-and-motion studies decompose actions down to the level of individual body movements.

Once a goal is decomposed into actions, the different actions can be assigned to actors. This assignment may be done in a hierarchical fashion; a high-level subgoal may be assigned to some organizational subunit, which further decomposes the goal and assigns particular actions to individual members of the subunit. Subgoals held by different parts of the organization may sometimes conflict. For example, design engineers may want to improve the quality of the product by fixing every problem, but plant engineers may want to optimize the performance of the plant by minimizing the number of changes introduced. In addition, individuals may hold individual goals which are

unrelated to (and possibly conflict with) the organizational goals. I will not consider these individual goals in more detail in this thesis, but in general it is obviously important to consider how individual and organizational goals interact.

5.1 Where do goals come from?

Most planning systems have been "one-shot"; the goal state is given and the problem is simply to develop a plan to achieve the goal. For my models, an actor's goals are not fixed; rather, actors may adopt new goals based on communication with other actors or in response to situations in the world. For example, asking an actor to do something gives that actor another goal. In the examples from the cases, I model the customer's request to the response center as giving the response center a goal of finding a solution to the problem.

However, I do not consider how the actors acquire their initial set of goals; rather, I simply model what they are. Wilensky (1983) has done some work in this regard that could be incorporated in a more complete model. He describes the implementation of a continuous planning system with a goal detector, which is responsible for determining that the system has a goal by recognizing situations requiring some action. The goal detector operates through a mechanism Wilensky calls a noticer, which looks for relevant changes in the world or in hypothesized worlds. When a particular situation is found, it triggers the introduction of a goal. Wilensky notes that these situation-goal pairs are similar to Schank and Abelson's (Schank and Abelson, 1977) themes. For example, a role theme causes an actor to have a goal of performing some task (like grading exams) by virtue of filling a role which requires it to do such tasks (such as the role as a professor) (p. 24). This particular situation is very general; more specific patterns may immediately invoke relevant default plans. For example, if a student wants to arrange a meeting, the planner for a professor may immediately suggest planning to meet the student during office hours.

6 Modelling actions

Actors achieve their goals by performing *actions* to change the state of the world. For example, the action `put-on(a, b)` results in a state of the world where block a is on block b `(on(a, b))`; `give(actor1, actor2, object)`, the state where $actor_2$ has the object. As in these examples, actions may be generalized action schemas with parameters; e.g., `put-on(Block1, Block2)` is the class of actions of putting some block ($Block_1$) on some other block ($Block_2$). Specifying the values for the parameters results in an action instance; e.g., `put-on(a,b)` is the specific action of putting block `a` on block `b`.

6.1 Preconditions

Actions may have preconditions, states of the world that must hold before the action can be performed.

Physical preconditions. Some preconditions are physical. For example, before a block can be moved by itself, there must be no other blocks on top of it; to give an actor an object, the performer must initially have the object.

Knowledge preconditions. Actions can additionally have knowledge preconditions (Moore, 1979; Morgenstern, 1988), that is, actors must know what actions to take before they can act. For complex actions, this knowledge includes knowing a decomposition of the complex action into primitive, directly executable actions. Since an action may have parameters, actors must know the parameters to perform even primitive actions. Moore (1979) gives the example of an agent frustrated in its attempts to open a safe by dialling the combination because it does not know the combination (or which safe a combination opens).

Skill preconditions (e.g., being able to do something, like read or play piano) can be treated as part of the physical preconditions for the action, e.g., a precondition for applying an action `read` is that the agent satisfies the predicate `reads` (Moore, 1979, p. 80). We may assume that only particular actors know how to perform a given action; for example, the response center can lookup a solution to the problem while the customer can not.

Strictly speaking, the knowledge preconditions for an action must be met only for the action to be done deliberately. For example, an actor with a goal to go to the largest city in the United States might not know that the largest city is New York, but might go to New York for some other reason (Davis, 1990, p. 418). In my study, I focus on deliberate actions; I assume that the knowledge preconditions must be satisfied before an action can be taken.

Using the knowledge preconditions for an action, an actor can reason about actions that serve as tests, like using a piece of litmus paper, from knowledge of the properties of the action (litmus paper changes colour in acids or bases) and from a knowledge of the outcome.

Social preconditions. Finally, actions may have social preconditions, meaning certain social conventions that must hold before an action can take place. For example, only particular actors are allowed to perform particular actions; other actions need to be approved before they can be performed.

6.2 The frame problem

One problem with this formulation of actions is what is called the frame problem. Actions change some aspects of the world (e.g., `put-on(a, b)` changes the location of block a) but other aspects of the world are presumably unaffected (e.g., the location of block b is not changed, nor, for that matter, is the location of the Eiffel Tower). In general, if actions are mappings from one state to another, each action must specify what happens to every possible fact in the world, even seemingly irrelevant ones.

One approach to this problem is the one adopted by STRIPS (Fikes and Nilsson, 1971). STRIPS represents the state of the world as sets of formulas and actions as sets of *preconditions, adds* and *deletes.* Preconditions are formulas that must be true for the action to take place (e.g., in order to put one block on top of another, `put-on(A, B)`, the tops of both blocks must be clear, `clear(A)` $\wedge$ `clear(B)`); adds are formulas that become true as a result of execution of the action (e.g., the first block is on top of the second block, `on(A, B)`); and deletes are formulas that become false, that is, that are removed from the set of true

statements (e.g., the second block is no longer clear, `~clear(B)`). STRIPS solves the frame problems by assuming that everything not mentioned by an action does not change. This representation of actions greatly simplifies the problem of planning a sequence of events to achieve a goal, which can be done by backward chaining from the preconditions of actions to effects.

In my models, I use the STRIPS approach and represent actions as sets of preconditions, adds and deletes. For example, the `put-on` action discussed above is represented as:

put-on(A, B)	
Preconditions:	clear(A) $\wedge$ clear(B)
Adds:	on(A, B)
Deletes:	clear(B), $\forall X \neq B$: on(A, X)

The `give` action is represented as:

give(actor$_1$, actor$_2$, object)	
Precondition:	have(actor$_1$, object)
Add:	have(actor$_2$, object)
Delete:	have(actor$_1$, object)

In many cases, the actions I discuss only add to the world and do not delete.

6.3 Primitive actions

Some actions are primitive and all actors know how to perform them directly. I write:

```
can(actor, action)
```

to mean that the actor can perform the action.

```
achieves-goal(actor, goal)
```

is true if there is some action the actor `can` perform and the action achieves the stated goal.

In order to focus on the coordination recipes, I assume two sets of primitives actions. These are actions I assume the actors can do but which are not in and of themselves coordination: (1) generic processes required for group action, such as communication and group decision making and (2) domain specific actions.

Generic processes are shown in Table 2.1 (adapted from Malone and Crowston (1990)). These layers are analogous to abstraction levels in other systems, such as protocol layers for network communications.

For instance, most coordination processes require that some decision be made and accepted by a group (e.g., which actors will perform which actions). Group decisions, in turn, require members of the group to communicate in some form about the goals to be achieved, the alternatives being considered, the evaluations of these alternatives, and the choices that are made. This communication requires messages be transported from senders to receivers in a language that is understandable to both. Communication actions are discussed in the next section.

Finally, the establishment of this common language and the transportation of messages depends, ultimately, on the ability of actors to perceive common objects, such as physical objects in a shared situation or information in a shared database.

6.3.1 Representing actions as capabilities

Most task knowledge in my models is expressed abstractly as the ability for certain actors to transform some input state into an output state. For example, software engineers are modelled as being able to determine which module has the problem (the output state) given a set of symptoms (the input state).

Knowledge and capabilities are in some sense interchangeable as a representation. Instead of simply saying that an actor can perform some task, we could, in principle, work out in detail the knowledge necessary to do it. In fact, knowledge engineers do exactly this when they develop an expert system. For the purpose of these models, however, such detail is usually unnecessary. It is important to know, for example, that software engineers can locate problems in particular modules (and that other actors can not); it is not essential to know in detail how they do that. Representing this bit of task knowledge as a capability greatly simplifies the development and representation of the model. Alternately

Table 2.1. *Processes underlying coordination.*

Process Level	*Components*	*Examples of Generic Processes*
Coordination	goals, actions, actors, resources, dependencies	identifying goals, ordering actions, assigning actions to actors, allocating resources, synchronizing actions
Group decision-making	goals, actors, alternatives, evaluations, choices	proposing alternatives, evaluating alternatives, making choices (e.g., by authority, consensus, or voting)
Communication	senders, receivers, messages, languages	establishing common languages, selecting receiver (routing), transporting message (delivering)
Perception of common objects	actors, objects	seeing same physical objects, accessing shared databases

stated, I have chosen a rich ontology of primitive actions because I do not wish to provide a full set of axiomatization for them (cf. Morgenstern, 1988, p. 8).

Furthermore, capabilities are an important part of the models actors have of each other. In Site A, for example, Marketing engineers know that software engineers can locate problems in modules, but they do not know in any detail how this is done. Nevertheless, they can reason about how to take advantage of this ability.

6.4 Complex actions

Primitive actions can be grouped together to form complex actions. Composition operators include sequences, conditionals, while loops, and concurrent actions (Morgenstern, 1988, pp. 127–130). These composition operators are represented as:

$$\texttt{sequence}(\texttt{action}_1, \texttt{action}_2),$$

meaning $\texttt{action}_1$ followed by $\texttt{action}_2$,

```
cond(condition, true-action, false-action),
```

meaning `true-action` if `condition` is true, otherwise `false-action`,

```
while(condition, action),
```

meaning, nothing (the null action) if `condition` is false, otherwise `action` followed by another while loop, and

$$\texttt{concurrent}(\texttt{action}_1, \texttt{action}_2),$$

meaning $\texttt{action}_1$ and $\texttt{action}_2$ starting at the same time; the complex action ends when the longer of the two actions ends. Versions of `sequence` and `concurrent` can be defined for more than two actions.

Because the organization is composed of multiple actors, multiple actions may happen simultaneously. For example, an engineer may start a designer

working on the detailed design for some part while soliciting comments from the other actors likely to be affected by the change. I model this process by assuming that actors can send messages and continue without waiting for a reply. However, I assume that a single actor, working alone, could perform only a single action at a time.

Knowing how to do a complex action means knowing a decomposition of that action into primitive actions. For example, Morgenstern (1988, p. 139) gives the example of knowing the definition of sharpening a pencil, presented here is a slightly modified format:

```
know(actor, equal(sharpen(Pencil, Sharpener),
    sequence(put-in(Pencil, Sharpener),
  while(~sharp(Pencil), rotate(Pencil)))))
```

Given a goal, actors must determine what composition of primitive actions to apply to achieve it. I do not discuss how this planning works; rather, I assume that given an achievable goal and knowledge of the available actions, the actor can develop a plan that achieves the goal, that is, actors are assumed to have an action plan:

```
plan(Situation, Goal)
Preconditions: know(Actor, Actions)
Adds:          know(Actor, Plan) ∧ achieves-
               goal(Plan, Goal, Situation)
```

There is, however, quite a lot of work on planning that could be included in a more complete model.

6.5 Where do capabilities come from?

Some actions are primitive and all actors are assumed to be able to perform them. For example, I will assume that all actors can communicate with other actors. Other actions may be restricted to a particular group of actors. The set of actions an actor may take depends on the actor's role as well as on its

physical capabilities or knowledge. For example, as discussed below, only particular actors are allowed to approve changes.

For complex action, actors must additionally know a decomposition for the action. Actors may reuse scripts or stored plans rather than planning from scratch each time they are faced with a problem. Actors may have developed and stored these plans or they may have been given them by someone else, e.g., during initial training in the job. When a stored plan fails, the actor may attempt to revise them, changing what they do to fit the changing circumstance. Alternatively, actors may be modelled as reactive planners: given a goal and a situation, actors check their memory for an appropriate action or set of actions to be performed, repeatedly performing actions until the goal is achieved.

7 Modelling communication between actors

Some of the facts known by an actor come from direct observation of the world or from prior experience. Many facts and goals, however, come from communications with other actors, such as the communication between the customer and the response center or between different software engineers. I represent the fact that one agent can talk to another by:

$$\texttt{can}(\texttt{agent}_1,\ \texttt{talk-to}(\texttt{agent}_1,\ \texttt{agent}_2))$$

Communication between agents can be included in a planning framework. Cohen and Perrault (1979) hypothesize that people maintain as part of their models of the world symbolic descriptions of the world models of other people and talk in order to take some action on these models. They suggest treating speech acts (Austin, 1962; Searle, 1969) as actions an agent can perform to affect their listeners' beliefs and goals.

There are five different kinds of speech acts: (1) declarative, expressing a fact; (2) interrogative, asking a question; (3) imperatives, telling a listener to do something; (4) exclamatory, expressing emotions, and (5) performative, speech acts that directly achieve some condition, such as pronouncing a couple husband and wife (Davis, 1990, p. 440). For my models, I need to handle all but

exclamatory speech acts. Cohen and Perrault (1979) define actions for declarative, imperative and interrogative speech acts. I present these operations here in a somewhat modified form.

7.1 Declarative speech acts

Informing an actor of the truth of some proposition `prop` is represented by the action `inform`. Note that this formalization does not allow for lying.

```
inform(Speaker, Hearer, Prop)
Preconditions: know(Speaker, Prop)
Adds:          know(Hearer, know(Speaker, Prop))
```

This model assumes that the hearer understands the request from the speaker, or, alternately, that there are no language problems. To model problems caused by differences in language, this action could be replaced by a more complete model that included the speaker's translating a goal into a statement in some external language and the hearer's interpretation of that statement. However, for the kinds of organizations studied in this thesis, this model seems adequate.

This action results only in the hearer knowing that the speaker knows something; to actually have the speaker know it also requires another step which Cohen and Perrault (1979) called `convince`. They model convincing quite simply by saying that if an actor knows that someone else knows something, then it knows it too:

```
convince(Speaker, Hearer, Prop)
Preconditions: know(Hearer, know(Speaker, Prop))
Adds:          know(Hearer, Prop)
```

This action embodies what is sometimes called the principle of charity: if an actor knows something, then that something is likely true,

$$\texttt{know(Actor, } \varphi\texttt{)} \Rightarrow \varphi.$$

Note that for knowledge this principle is true by definition, but for belief, it is only a plausible inference (Davis, 1990, p. 362). Cohen and Perrault point out that a better formulation would requires the `speaker` to convince the `hearer` that `prop` is true by providing justifications which are eventually grounded in mutually held beliefs. For my purposes, however, this model of convincing is sufficient.

7.2 Imperative speech acts

A request from one agent `speaker` to another `hearer` to do an action `act` is modelled by a `request` action.

```
request(Speaker, Hearer, Act)
Preconditions: know(Speaker, can-do(Hearer, Act )
Adds:          know(Hearer, want(Speaker, Act))
```

Note that this action only results in the hearer's believing that the speaker wants the action. To cause the hearer to also want the action, Cohen and Perrault (1979) hypothesize a step named `cause-to-want` which models what is required to convince someone to want something. They model this process very simply: to get someone to want to do something they can do, one need only tell that person that you want them to do it.

```
cause-to-want(Speaker, Hearer, Act)
Preconditions: know(Hearer, want(Speaker, Act)) ∧
               know(Hearer, can(Hearer, Act))
Adds:          want(Hearer, Act)
```

This model of motivation is unrealistic, but I believe it is sufficient to model the organizations I studied. To model organizations where different actors have different goals would require a better model of what it takes to persuade actors to do something. However, those changes would mostly take the form of additional preconditions to this action and could thus be handled by this mechanism.

7.3 Interrogative speech acts

The `inform` action allows an actor to say that it knows something. However, by itself this action is not very useful for asking questions (that is, for requesting that an actor perform an `inform` action), since planning for someone to execute an `inform` action requires knowing in advance what will be said. Therefore, there must be another action whose performance can be requested to ask questions where the speaker does not know the answer to the question but can only provide a description of the answer.

Cohen and Perrault (1979) handle this case with a new action `informref`, which is given a predicate of one variable. The information requested is an object which makes the predicate true. For example, if an actor asks "Where is Tom?", it wants the value of `X` that makes `location(tom, X)` true. In this formalization, `λX: d(X)` means a function taking one variable `X` and returning the value of the predicate `d(X)`, either true or false; `ιX: d(X)` means the value `X` for which `d(X)` is true. `informref` is defined as:

informref(Speaker, Hearer, λX: d(X))	
Preconditions:	∃Y: know(Speaker, ιX: d(X) = Y)
Adds:	∃Y: know(Hearer, know(Speaker, ιX: d(X) = Y))

Using this action a speaker can `request` that a hearer perform an `informref` action, providing a description of the desired object in the form of a predicate to be satisfied. For example, to ask "Where is Tom?", the speaker performs the following action:

```
request(speaker, hearer, informref(hearer,
        speaker, λX: loc(tom, X)))
```

When the hearer performs the requested `informref` action, the speaker then knows the object that hearer knows satisfies the predicate. To complete the performance of this action requires a new mediating action, `convinceref`:

```
convinceref(Speaker, Hearer, λX: d(X))
Preconditions: ∃Y: know(Hearer, know(Speaker,
               ιX: d(X) = Y)
Adds:          ∃Y: know(Hearer, ιX: d(X) = Y)
```

Cohen and Perrault (1979) define a similar pair of actions for asking yes/no questions, where the communicated value is a simple Boolean.

7.4 Performative speech acts

The final class of speech acts are performatives, where a speech act results in some action due to some agreement between actors. For example, the speech act of pronouncing a couple husband and wife itself changes the state of the world, due to social convention (Davis, 1990, p. 440). Performative speech acts are the way actors satisfy the social preconditions of actions.

For my study, I need to model occasions where one actor must have some proposed action approved by another actor. In these cases, the approving actor does not actually "do" anything more than sign a form; but the process can not continue without this approval. I model such approval process by assuming that certain actors could assert that a particular object, such as a change, had been approved:

```
approve-change(Performer, Change)
Adds:          approved-by(Performer, Change)
```

Only certain other actors can perform this action and change this state. Other actions then have:

```
approved-by(manager, Change)
```

as a precondition.

3

Study Design

...accurate description and verification are not so crucial when one's purpose is to generate theory... evidence and testing never destroy a theory... they only modify it.

—Glaser and Strauss, *The Discovery of Grounded Theory*

In order to informally test the power of my modelling technique and to fill out my typology of coordination methods, I used it to study a coordination-intensive task performed in three large companies. In this chapter, I describe this field study. I first discuss the implications of my choice of research methodology, namely case studies. I then present the particular task I chose to study, engineering change management, and the sites in which I studied it. I describe what data I collected at these sites and how I collected and verified it. I conclude by outlining the steps I follow to develop a model of the organization from the data collected, including an example of the analysis of a brief section of an interview.

1 Case study methodology

A research question can be studied using many different research methodologies. Each methodology has unique strengths and weaknesses, making it more or less useful in different circumstances. My goal in this thesis is the development of a theory of how coordination works in organizations and a technique for studying it. I approach these questions through the use of case studies of a coordination-intensive process in human organizations. I chose case studies because they provide rich empirical data, necessary for theory generation.

Newell and Simon (1972) use cases studies of individuals solving crypt-arithmetic problems to suggest how humans process information. In their cases, they use the performance of an individual on a task to develop a model of the information processing involved, resulting in a theory of individual problem solving. I similarly generate a theory of coordination by modelling the way individuals in organizations coordinate a particular task.

In the postscript to their book, March and Simon note questions about the status of field research and single case studies of an organization (p. 12). Since they wrote, however, researchers have better defined the utility of this research paradigm. In particular, Yin (1984) notes that case studies are particularly appropriate for answering "how" or "why" questions about current events in situations where the researcher has no control over the circumstances of the study.

For answering specific questions about "how many" or "how often," where the goal is to be predictive, survey methodologies usually offer better coverage of the study population and greater external validity. For the current study, however, I want to investigate the mechanisms through which coordination is being provided and the circumstances that make those mechanisms appropriate.

If the study concerns past events, some form of archival research or historical research is more desirable. In this thesis, however, I study coordination as it was currently provided.

Finally, if researcher has control over the events of the study, an experimental methodology may be more appropriate since it assures better internal validity. I am interested in coordination in large organizations, however, where I have little or no control over events. Future studies may involve an intervention in the organization in the form of a computer system designed to help coordinate. In this case, a quasi-experimental design may be more appropriate.

One methodological issue affecting the choice of case studies is their potential lack of external validity. It is often difficult to tell how much can be generalized from any particular case study. In this study, I address this concern by studying multiple case sites and comparing my findings across them. It should be noted, however, that my goal is mostly to suggest a new framework for thinking about organizations; as such, I am not suggesting that all organizations work the way I describe.

Another issue concerning a case study methodology is what intellectual framework will discipline the selection and interpretation of observations. I address this issue with the modelling technique described in Chapter 2. In some cases, it may not be possible to fully specify the models, but attempting to do so provides a check on the completeness and consistency of the information collected. As Cyert and March (1963) note:

> The likelihood that a process model will incorrectly describe the world is high, because it makes some strong assertions about the nature of the world. There are various degrees by which any model can fail to describe the world, however, so it is meaningful to say that some models are more adequate descriptions of the world than others... that the agreement cannot be attributed to mere coincidence. (p. 319).

The process of developing a model for an organization is discussed below.

2 Task selection

I this section I will describe the particular task I chose to study, namely, engineering change processing and my reasons for choosing it.

In designing my study, I want to ensure that the cases selected can be meaningfully compared and yet are different enough to suggest an interesting range of coordination mechanisms. This goal places several constraints on my selection of cases.

First, because my definition of coordination depends on the goals of the organization being studied, I held the task constant. Allowing the task to vary might result in a greater variety of coordination processes, but it would be much

more difficult to isolate the reasons for the differences observed. Second, studying the same task results in some economies for the researcher, since the processes have some similarities.

Second, selecting a single task keeps the level of aggregation constant, avoiding that limitation of my definition of coordiation. Examining a single process provides a clear definition of the subset of relevant people to study. McGrath (1984) defines a group as having, "interaction, interdependence, mutual awareness, a past and an anticipated future"(p. 6). The groups I studied are long-term, limited-band natural groups, which he calls crews or work teams (p. 44), except they may be larger, with the result that members of the group may not know all other members.

2.1 About engineering changes

The process I chose to study is the engineering change management process, that is, the process used by companies which design and manufacture products to control changes made to the product's design.

A product design process typically goes though several stages. In the earliest stages, the design is worked on mostly by members of the engineering department. In order for the product to be produced, of course, other groups, such as purchasing or manufacturing, must be involved. At some point, therefore, the design is *released,* which means that design documents describing the product are made available to the other groups. For my thesis, I focused on changes made to the design after it had been released. I did not address changes made to the manufacturing processes, tooling, or any of the many other aspects of the production process, or changes made during the development of a new design before the initial release.

Engineering changes include three distinct kinds of changes: (1) corrective: fixing errors where the system does the wrong thing, e.g., a bug in some calculation in a computer program; (2) adaptive: making the system do something different in response to changing needs, e.g., changing calculations to conform to new tax laws or adding new functionality; and (3) perfective: making

the system do the same things better, e.g., increasing performance (Lientz and Swanson, 1980). We can also distinguish between changes made in response to new needs and those made in response to changes in the underlying system (e.g., new hardware or a new version of an operating system).

In different industries different kinds of changes may be common. For example, most companies need some process to make corrective changes to fix unanticipated design defects. However, companies differ in how they manage adaptive or perfective changes. In some companies these other kinds of change are addressed primarily by designing a new product, or at least a new version of the product. For example, car companies make adaptive and perfective changes primarily between model years of a car. Other companies, such as the commercial jet aircraft company, introduce new versions of their products only rarely; in these companies, all three kinds of change are handled by the same process.

2.2 Parts

Complex products of the sort I studied are assembled from simpler component parts. Parts vary in complexity. Some are simple items, such as a moulded piece of plastic, a pressed sheet metal body panel or a standard nut or bolt. Others are complete subassemblies, such as a radio or even an entire engine. For Car Co., one of my case sites, a part is the smallest unit designed by a Car Co. engineer and assigned a Car Co. part number. Two parts are the same and have the same part number if they have the same fit, form and function, meaning that they can be interchanged without effect. (Fit means the parts fit the same attachments; form means the parts fit within the same design space; and function means the parts meet the same design specifications. Note that the parts need not be completely identical to be interchangeable.) If the parts can not be freely interchanged, then they are different and must be assigned different part numbers. The part numbers are important, since the use of a part can be tracked only by part number.

Clark (1989) distinguished between three categories of parts: (1) detailed control parts, where the automobile company does most of the design and detailed engineering and the supplier acts mostly as a source of manufacturing capacity; (2) black box parts, where the automobile company supplies a functional specification and the supplier does most of the detailed engineering and manufacturing; and (3) off-the-shelf parts, where the supplier does the basic design and engineering for a part and the automobile company buys them, perhaps with some modifications.

2.3 Stages in implementing engineering changes

In order to bound the process somewhat and make the study feasible, I have chosen to focus on the design engineer, or more precisely, on the information processing done by the design engineer.

This focus suggests a tri-part division of the change management process. First, the engineer determines that some kind of change is necessary, e.g., by receiving a report of some problem. Second, the engineer develops a change, verifies that it is acceptable and has it formally approved. Finally, the change is implemented and the product modified, e.g., by the production plant after the engineer releases a new set of drawings. In my study I focused primarily on the first and second of these three steps.

Obviously these subtasks interact. For example, the organization of the design engineers in the second subtask dictates how a problem report must be routed to achieve the first subtask. Nevertheless, examining each subtask independently does have some advantages. First, if we can identify a subtask as an example of some generic coordination task, we can begin to analyze it and suggest alternative ways this task could be organized. Second, it provides a structure for comparisons of the process in different companies.

2.4 Engineering change control

Given that changes to the design will be necessary, nearly all companies choose to implement a change control process to control the changes. Typically

what are controlled are the documents describing the product, such as drawings or interface specifications. The product is supposed to match the documents describing it (the production processes may go to great lengths to ensure this), so if the documents do not change, the product should not either.

The purpose of a change control process is, as one interviewee said, to prevent changes to the product. More accurately, the process is intended to ensure that only good (e.g., necessary, well thought out, cost-effective, etc.) changes are made. Change control is typically imposed only after the product is stable enough that frequent changes should not be necessary and other divisions have begun to work with the design, increasing the cost of a change. Often, however, the decision to start the change control process is as much a result of timing as of any intrinsic property of the product. The products I studied were already subject to change control at the phase I studied them.

There are several reasons for having a change control process.

Approval. One important reason is to record the formal approval of the change by the engineer. Engineers are legally responsible for their designs, and all changes to those designs must therefore also be approved by them.

Ensuring design is consistent with engineering intent. A key problem for engineering is ensuring that a product can be reproduced as designed. The design as documented must stand by itself, since the design engineer will not be present to explain it. A goal of a change control process is to ensure that all changes made to the design are accurately and unambiguously reflected in the documentation and that all changes made are intentional. All stages of the design must be similarly controlled. For example, the design process of an automobile must document what parts were used in each car in case of a recall.

Informing users of the design. Any changes made by an engineer need to be made visible to the downstream group who will implement them. Many other aspects of the assembly process may need to change at the same time as the design. For example, someone must ensure that any necessary new parts,

assembly processes or tools will be ready at the same time. This is only possible if all groups are properly informed about any changes.

Controlling costs. A final reason for a change control process is to control the costs of changes made. Changes can be expensive in many ways. Some costs are direct, such as the cost of an engineer's time to redesign a part, of new tooling or the obsolescence of existing parts. Others are indirect, such as the reduction in the productivity of the assembly plant caused by any change in the process. In some cases, the costs outweigh the benefits of the change. Each change requires a business decision to ensure that the costs are justified. Often this involves an additional approval process, since there is often a perception that engineers are more interested in getting a perfect design than one that is cost-effective and therefore introduce unjustifiable changes to improve their design.

2.5 Why study change management?

I believe the engineering change process provides an instructive microcosm of organizational coordination issues and is well suited for my study for several reasons.

First, the process is one that requires coordination. For simple products, this process may be very simple; for example, if the product is made by a single person, he or she may simply make the next one a little differently. If the design and manufacturing are done by different groups, however, coordination is required between the designer and the manufacturer to communicate the desired changes. For very complex products, there may be hundreds of designers and many groups affected by a change in the design. In these cases, the change control process may require a very large amount of coordination. Even within the same firm, different parts seem to require different kinds of coordination. For example, interviewees in several companies have commented on the difficulty of managing changes to electrical wiring and hydraulic piping, components that often link systems designed by different groups.

Second, the process is one that could be supported more effectively with an information system. For instance, the Information Lens (Malone, et al., 1987)

or Object Lens (Lai, et al., 1988) system might be useful, since much of the data exchanged is (at least) semi-structured and subject to fairly straightforward processing rules. In fact, many companies seem to be implementing computer systems to support engineering data management (e.g., "Engineering made easy", 1990).

Third, the restricted nature of my model suggests studying situations where the processing is done in a routine fashion, to reduce issues of interpretation by the members of the organization. Managing engineering changes is primarily an information-processing task, often with developed formal procedures (at least once a change has been developed), and the goals of the individuals working on it seem fairly clear.

Fourth, the process is significant in many manufacturing industries and is one that many companies feel they could do better. Reducing the need for changes and the cost of making changes may be key to improving the flexibility and responsiveness of manufacturing organizations. Companies spend millions of dollars making changes to their products, so even small improvements in the process could result in large cost savings.

Finally, there seems to be surprisingly little prior research on this topic. For example, a quick survey of books on engineering management (e.g., Coxe, 1980; Hilton, 1985; Leech, 1972) revealed that most covered the topic in a page or less.

3 Case site selection

In this section I will discuss my criteria for choosing research sites in which to study engineering change processing and briefly describe the companies I have selected.

The first question is how many sites to study. I chose to study only three sites, for two reasons. First, I am mostly concerned with generating interesting ideas about coordination processes and demonstrating my research methodology. I am not attempting to prove or disprove a particular hypothesis,

which would require a much larger sample size for conclusions to be statistically significant. Second, focusing on a few companies allows each one to be examined in greater detail.

It is important to note that I am not systematically surveying all possible organizations. Rather, I focus on a few sites with an interesting mix of products, production techniques and change processes, a technique sometimes referred to as theoretical sampling. This approach somewhat limits the range of coordination mechanisms I expect to observe. In particular, I do not expect to find organizations with completely unsuccessful mechanisms, because such organizations would have to quickly change or go out of business. I have no particular reason to suspect that the sites I chose are atypical. Nevertheless, this selection technique does suggest caution in generalizing from my findings and highlights the need for more systematic studies to confirm my results.

I applied several criteria to my selection of case sites.

Complex products. I want to study companies that make complex products, for two reasons. First, the complexity of the products implies a need for many engineers and downstream groups, so the coordination cannot be done entirely in face-to-face meetings. Second, I want products that require changes with non-local effects. I started my search by listing complex products—such as cars, computers and jet aircraft—and companies that manufactured them. As another way to locate appropriate companies, I looked for companies with many engineers and high spending on research and development ("R & D Scoreboard", 1988, p. 140).

Interested companies. Obviously, an important pragmatic requirement is to find companies that are interested in the research. This criterion may introduce a selection bias, since companies interested in the study might also be more active adopters of new management techniques. Companies that feel their change management processes work badly might be unwilling to be studied and in fact that seems to be the reason one potential site chose not to participate.

Variety of industries. In order to introduce some variety into the engineering and production processes examined, I studied organizations in a variety of industries. Varying the industry gives the case sites a range of organizational structures and environmental conditions.

Different need for changes. One aspect that varies is the need for engineering changes. For some products, eliminating changes is a reasonable goal, since most unplanned changes are implemented to fix engineering errors or problems in the manufacturing process. In other industries, however, products must be extensively tailored for each customer, making a certain number of changes inevitable. In most industries, the product is constantly being improved, but these improvements are introduced in different cycles.

Different production processes. Another way the sites differ is in production process. Some products, such as automobiles, are mass produced on assembly lines. Others, such as commercial aircraft, are custom built in small batches. Computer software is a one-of-a-kind design effort, with relatively small production costs (e.g., Lotus has a policy that no product should cost more than $5 to manufacture[1]). These production processes affect the difficulty of making changes. For example, making a change to a computer program is relatively straightforward, once the problem is identified; making even a minor change to an automobile may require weeks of retooling.

Different degree of control of changes. Finally, organizations differ in the extent to which they can control their work flows. According to Ashby's Law of Requisite Variety[2], an organization needs one policy lever for each environmentally determined variable. Therefore, groups in environments with many exogenous variables should have more developed ways of controlling for those variables. For example, car companies determine the timing of changes themselves; aircraft manufacturers, by contrast, are subject to a constant demand

---

1 Personal communication from a former employee.

2 This was suggested to me by Mike Epstein.

for changes from their customers. This law suggests that aircraft manufacturers need more developed mechanisms to control the flow of change requests.

Perceived problems with changes. Although I do not use it as a selection criteria, some sites have greater perceived problems with their change control processes. For example, a common perception was that too many changes are being made. Also, some of the organizations are experiencing additional stresses that affect their ability to manage changes. For example, some have recently reorganized their engineering groups; in one, the experience level of the engineers has dropped significantly. I believe these cases can be especially enlightening, since they suggest more dramatically what coordination is necessary, as well as showing how increased production can substitute for the missing coordination.

3.1 Brief summary of sites

3.1.1 Site A: Computer Systems Co.

Site A is computer company at which I studied the design of computer systems software, such as an operating system and database system. The field work at this site was carried out by me and by another doctoral student, Steven Brobst.

The software design group is relatively small, so it is possible to coordinate the activities of the group informally. Individuals simply remember who uses the modules they wrote and when they need to make a change, talk directly to those people. The size of the group is growing and groups in other sites are starting to use the software, however, increasing the difficulty of coordinating changes. When we visited, a system was being planned that would automate some of the tracking of users that had formerly been done by the engineers. Also, the organization underwent several reorganizations while we were studying them, some of which appear to be linked to changes in the product environment.

3.1.2 Site B: Car Co.

Site B is a division of a large automobile manufacturer. This division designs several models of expensive (about $35K) cars built in relatively small numbers (200,000 to 300,000 per year). The field work at this site included about 10 days of interviews of people in downstream organizations plus about 5 days of observation of engineers at work.

The engineering group in this site involves hundreds of engineers, divided into seven functional groups. Engineers are largely responsible for determining what changes they want to make. These changes then have to be approved by the managers of the engineering group. Once a change is approved, a fairly elaborate change control process ensures that the downstream groups implement "engineering-intent." Coordination between different engineering groups seems to be mostly informal or through physical models of the car.

3.1.3 Site C: Airplanes, Inc.

Site C is a manufacturer of commercial aircraft. I spent about sixteen days at this site. Some of this time was spent becoming familiar with organization and some in interviews with people involved in the change control process. I also observed the meetings of three different change control groups.

The engineering group involves several hundred engineers divided by function, e.g., payload, structures and propulsion. Engineers again determined what changes they want to make. In addition, customers request changes which require additional design effort. Changes need the approval of both the chief project engineer and of the "Superboard" which included most of the top management of the company. A group within engineering, called Engineering Change Control, oversees the progress of changes from an engineer through the approval process and to manufacturing, who are responsible for implementing it.

4 Data collection

In this section, I describe the way I collect data about engineering changes at my field sites. I attempt to uncover, in March and Simon's (1958) terms, the

programs used by the individuals in the group. March and Simon suggest three ways to uncover these programs: (1) interviewing individuals, (2) examining documents that describe standard operating procedures or (3) observing individuals. I was able to use all three processes in my study.

Data collection starts by identifying the different types of actors in the group being studied. Identification of the different types of groups in the organization is done with the aid of a few key informants, and refined as the interviews progressed.

I then identify the information received by each kind of actor and the way each type of message is handled. Information about the kinds and sources of messages received, the way they are processed and the recipients of messages sent is primarily obtained from interviews with individuals in each group. This information is summarized in an information-flow model of the organization.

It is interesting to note that the process found frequently differs from the formally documented process. For example, at one site, engineers receive a listing of all approved changes, but the official list seems merely to confirm that the changes have been approved. In order to react to a change, the engineer must be warned of it well in advance of its appearance on the official list. This warning seems to happen primarily through an informal process. It is this informal process I seek to document.

4.1 Data collected

I collected three major kinds of data: interview data, observational data and documents describing jobs.

4.1.1 Interview data

Most of the data collected for this study came from semi-structured interviews with various members of the organization. As March and Simon (1958) point out, "most programs are stored in the minds of the employees who carry them out, or in the minds of their superiors, subordinates or associates. For

many purposes, the simplest and most accurate way to discover what a person does is to ask him" (p. 142).

Data is collected by asking subjects: (1) what kinds of information they receive; (2) from whom they receive it; (3) how they receive it (e.g., from telephone calls, memos or computer systems); (4) how they process the different kinds of information; and (5) to whom they send messages as a result. (An outline of the questions asked is included in Appendix 1.) I behaviourally ground these questions by asking interviewees to talk about the events that have recently occurred and using those events as a basis for further questions. For example, I asked some individuals to go through their in-boxes and describe the different kinds of documents they found and what they do with each one (e.g., Brobst, et al., 1986; Malone, et al., 1987).

Interviews typically last a minimum of one hour and in some cases as long as three hours. In Appendix 2, I present an excerpt from an interview at Airplanes Inc., the third of my case sites.

Some researchers (Bernard, et al., 1985; Bernard, et al., 1980) show that individuals often forget some communications and over-remember others. These effects distort communication data gathered by simply asking interviewees who they talk to and how often. Freeman, Romney and Freeman (1987) show that this bias does exist but tends to emphasize long-term communications. As Ancona and Caldwell (1990) put it, "respondents are not actually answering the question 'Whom did I speak to in the last two weeks' but 'In a typical two week period, with whom am I likely to have spoken'." Since I am interested precisely in these long-term patterns, I chose the simpler method of directly asking people about their communications.

4.1.2 Other data

I also collected data about: (1) the types of information stored in computer systems; (2) the use of computer systems; (3) names on memo distribution lists; and (4) the kinds of forms used.

Another source of data is material that describes standard procedures or individual jobs. March and Simon (1958) suggest that these documents are created for three different reasons: (1) as instructions for the individuals doing the job, (2) as descriptions for new members of the group and (3) to legitimize or formalize the procedure (p. 142). They note that the interpretation of these documents depends on the purpose it was intended to serve.

4.1.3 *Observational data*

Finally, to get a better sense of the kinds of communication individuals actually use, I observed some individuals during the course of a typical work day. For example, in one site I tailed an engineer for a day, during which I sat in on scheduled and unscheduled meetings and took notes about the kinds of people the engineer interacted with and the types of information exchanged.

4.2 Validation of data

Relying on interviews for data can introduce some biases. First, people do not always say what they really think. Some interviews were conducted in the presence of another employee of the company, so interviewees may have been tempted to say what they think they should say (the "company line"), what they think I want to hear or what will make themselves or the company look best. Second, individuals sometimes may really not know the answer.

Some of these biases can be controlled by cross-checking reported data with other informants. For example, if one interviewee reports sending information to a particular group, I can check if that other group reports receiving such information.

Furthermore, the modelling process serves as another check on the consistency of the data. I used an iterative approach, sometimes called the negative case study method (Kidder, 1981), switching between data collection and model development. The initial round of data collection serves as the basis for an initial model. Constructing this model reveales omissions in the data, for example, places where it was not clear how an actor reacts to some message or

from whom a particular piece of information comes. These omissions or ambiguities serve as the basis for further data collection.

Despite possible problems with the data, I feel that this collection technique is appropriate for my study. I am interested in what is supposed to happen in the change process and I have no reason to believe that subjects were deceiving me about what they did. Furthermore, the goal of my data collection is not to document precisely how engineering change process works in these companies (although I believe I have done that) but rather to collect raw material for the development of a theory about coordination processes. In some sense, the validity of the particular data that serves as the basis for this theory is irrelevant to the validity of the theory itself. As Glazier and Strauss (1967) put it, "accurate description and verification are not so crucial when one's purpose is to generate theory... evidence and testing never destroy a theory... they only modify it" (p. 28).

5 Analysis technique

As discussed in Chapter 2, I seek to understand the organizations I study by constructing models of the behaviour of the actors in those organizations. The goal of this modelling is to describe the knowledge and capabilities of each actor (resulting in what I call intentional models, since they capture the intentions that lay behind the individuals' actions); however, as I describe in this section, I first abstract the information-processing behaviour of the individuals by constructing what I call information-flow models and then use those models as a basis for constructing the intentional models.

5.1 Information-flow models

I want a more succinct description of the communication that makes it clear what kinds of information is exchanged, where the information comes from and goes to, and how different kinds of actors process it. In an earlier study (Crowston, et al., 1987), I approached this problem by developing what I now call information-flow models. These models are similar to data-flow diagrams (see,

for example, Yourdon, 1989) or the structured analysis and design technique (see Marca and McGowan, 1988). Because the coordination is achieved through communication, the coordination mechanisms being used leaves particular patterns in the observed communication flow.

Information-flow models comprise two major elements: actors and messages. Actors send messages to other actors. When an actor receives a message, it takes some action, which may include sending additional messages to other actors. Each kind of actor understands and reacts to a different set of messages. I use the term "message" here in an abstract sense that includes any kind of communication, verbal as well as paper or electronic.

Messages may be passed in a variety of media and I do not differentiate between media in terms of the effect of the message. For example, engineers often communicate with each other in face-to-face meetings; they release information about a new part by filling out a form and delivering it to a specifications clerk; purchasing gets the new information as a change in a database record. These examples show the variety of media used, but all are interpreted as messages.

For each type of message, I determine how the actors processes the information and what kinds of messages (if any) they send in response. I then describe the communications links between actors and the kinds of messages that are sent on each link. This modelling is typically an iterative process: identifying the response of one actor to a message may lead to the identification of other message types or suggest new kinds of actors who are involved.

For example, there might be a number of individuals designing parts of a car, all working in roughly the same way and using the same kinds of information; each would be an example of an "engineering actor." Manufacturing actors use different information and would be analyzed separately. The engineering actors receive notifications of changes in other parts from other engineering actors or of problems in building their parts from manufacturing actors.

When engineering actors receive change notices, for example, they first determines if the change described might require changes to their parts, using their knowledge of the interactions between parts of the automobile. If it does, the engineering actors make the necessary changes. They then determine which other actors might be affected by their changes and send their own change notices to those actors and to the downstream organizations responsible for implementing the change.

The product of this analysis is a specification of the details of the types of messages and the behaviours of the actors, similar to a program written in an object-oriented language (e.g., Goldberg and Robson, 1983; Stefik and Bobrow, 1986). In principle, this model can be given in enough detail to construct a computational model of the organization. The object-oriented metaphor suggests creating a hierarchy of actor types as a way to simplify the description of different actors and to highlight their similarities; this method was used in Crowston et al. (1987). In presenting the model, I give only a description of the different types of actors (the classes) and the actions they take for each kind of message they understand; I do not present a full instantiation of the model. A working simulation would have an object of the appropriate class to represent each actor in the modelled organization.

5.1.1 Development of information-flow model

In this section I describe the process of creating an information-flow model from the sample interview shown in Appendix 2.

First, I identify the types of actors involved in the process. In the interview, the subject mentions several actors, including the customer, the customer engineer, business management, IE people in operations, the "lav." (lavatory) vendor and the Change Board.

Next, I determine what messages the actors send one another. I then specify what processing the actor does when it receives each type of message and in particular, what other kinds of messages it sends and to whom the messages are sent. In this example, the customer sends *requests* to the customer engineer

(lines 5–14 and 17–18); the customer engineer turns the requests into *CR proposals* and sends them to business management (lines 41–42); business management negotiates with IE (industrial engineering) for a date by which the changes can be committed and gets a line position (lines 117–120). The customer engineer then develops the *customer detailed specification* which is used to develop the statement of work (lines 167–169).

This brief excerpt leaves open many questions, such as: Who develops the statement of work (mentioned on lines 157 and 158)? Where does it go? Who else sees the CR proposal and the customer detailed specification? These questions would be addressed in further interviews until a relatively complete model can be prepared.

5.1.2 *Example of an information-flow model*

The resulting portion of the information-flow model is shown in Appendix 3. Note that this is only a fragment of the full model and includes data from other interviews and sources not discussed above. Details of the processing of some messages has been omitted; for example, information about Test Integration, Engineering Cost and Schedules and Weights has been left out. The full model is given in appendix to Chapter 6.

Each entry in the table shows the action taken by a particular actor in response to a particular type of message from a particular sender. Reading across, each line shows what type of actor sends the message, what message type is sent, what type of actor receives the message and what action is taken by the receiving actor. In some cases, although not in this example, the name of the message is shown in bold italics; this indicates that the message name is the same as a form actually used by the company studied.

The numbers on each line are not in any particular order; rather they serve to identify actor-message units. They are used, for example, in the actions to cross-reference where a message sent by one actor is processed by another. Several messages may have the same action, such as when an actor needs to

collect several messages before proceeding; in this case the action is shown once and the other entries in the table refer to that entry using the line number.

Where no sender is shown, the action is one taken spontaneously (or at least, for reasons outside the boundary of the model) by the actor. For example, on line 1, the model shows that the customer decides (for unmodelled reasons) that it needs some feature and sends a request for the feature to the customer engineer.

This excerpt shows the messages in roughly the order they are sent or received. The models in the rest of the thesis group all messages understood by a particular actor together and are in alphabetical order by actor.

The model shows (on line 2) that the customer engineer, in response to the message sent by the customer, prepares a *Change request work statement*, and sends a request to the project engineer to classify the change, that is, to indicate roughly how much work is involved in making the change. Once the project engineer responds (line 27) the customer engineer then distributes the change to various other groups for input and sends the change package to the Change Review Board.

The Change Review Board determines if the change can be offered in the time available (line 3), based on input from the various other groups (not shown). If it can not be offered, a *Change Request Rejected* message is sent to the customer. Otherwise, the change is sent to the customer as a *Proposal* message.

The customer then decides (line 15) whether or not to accept the proposal (i.e., if the change is worth the price requested) and sends the appropriate message to the contract group. If they accept (line 17), the contract group tells the customer engineer (line 19), who updates the Customer Configuration and informs the project engineer.

Currently, the models I have developed are represented simply as text. This technique is unsatisfactory for developing complex models, since it provides no support for, for example, checking the consistency of a model or actually executing the model as a simulation. Furthermore, the text is not very

perspicuous. Given the similarities of my modelling technique to existing structured analysis techniques, I am interested in possibility of using a CASE tool to help manage the models or perhaps developing a custom tool based on the Object Lens system (Lai, et al., 1988).

5.2 Intentional models

I next model what each actor must know in order to process messages in the way shown, using the modelling approach developed in Chapter 2.

5.2.1 Level of detail

In order to reason about the possible effects of their actions, actors need mental models of their actions and the things they act on in the world. To be complete, my models of the actor must include these mental models.

However, the actors' mental models can be constructed at different levels of abstraction. For example, in describing the processes involved in making a change, I say simply that the engineer develops a proposed solution to a problem. In more detail, this involves locating the problem, identifying its cause, thinking of ways to remove the cause, etc. Each step could be further decomposed, if desired; for example, locating the problem might involve recreating it, gathering more information about how often it occurs, etc.

There is nothing about any particular level of decomposition that makes it the correct level of analysis; rather, some level will be sufficient to model the particular phenomena in which we are interested. Picking an appropriate level is mostly a pragmatic decision. I want to be able to model at least the organizations observed, so tasks need to be decomposed at least to the level observed. For example, if we noticed that a designer was the one who actually prepared the revised drawings, we might want to treat that task separately. On the other hand, if we were primarily interested in the interactions between the engineering department and the rest of the company, we might be satisfied representing both departments as black box and ignore these steps. Some actions might be divided

even further, for example, to represent a division of labour that seems possible and interesting but which was not actually observed.

In principle I would like to describe these models in enough detail for a model of the actors simulated on a computer to actually solve the same problems that the actors do. For most real world cases, however, this implementation is difficult and time-consuming to do. Particularly for a thesis, there is a tradeoff between the time spent implementing the models and the time spent analyzing them.

In this thesis, I am mostly concerned about the interactions between the design engineer and the rest of the company. For this reason, I do not model much of the rest of the company. Furthermore, I am not attempting to model the entire thought processes of an individual engineer; I therefore model individual problem solving quite simply, going into detail only where the details have an effect on the engineer's interactions with other actors. In most cases, therefore, I will model the actors' domain knowledge and actions at a more abstract level and focus instead on the coordination processes.

For example, automobile engineers know a great deal about the structure of the particular car on which they are working, the production processes used to manufacture and install their parts, the qualities of the materials used, etc. In order to fully reproduce the communication necessary to solve a problem a model may require many details about the problem as well as as some difficult-to-characterize "common sense." My focus on coordination allows me to avoid some of these issues, since I can cut off the modelling process at the individual level and simply assume that the actor can solve the problem. However, in some cases a great deal of domain knowledge is necessary in order to be able to coordinate; for such cases, the models must be equally complex.

5.2.2 *Development of intentional models*

To develop a model for an actor, I first look at the actions the actor performs and describe them as some kind of transformation. For example, in the excerpt, the customer engineer can translate customer requests into change

request work statements (CRWS) (line 27 of the information-flow model); the change review board can determine if a proposed change can be offered (line 8 of the model). The intentional model for the actor then includes the ability to perform theses transformations. Again, actually performing the transformation requires a considerable amount of (mostly unmodelled) domain specific knowledge; the model merely states that the actor is capable of performing such an action. The models developed from the sample information-flow model are shown in Appendix 4.

The model for the customer engineer includes an action such as the following:

```
can(customer-engineer,
   develop-crws(customer-engineer, Change))
```

```
develop-crws(Performer, Change)
Effect:         ∃CRWS: have(Performer, CRWS) ∧
                change(CRWS) = Change
```

This is read as follows: the customer engineer can perform an action called `develop-crws`, given a `change` object as a parameter. Performing the `develop-crws` action has no preconditions (aside from knowing how to do it and knowing the change to which to do it) and has the effect that the actor performing the action has a `crws` object and that the `change` field of that object is equal to the original change.

The model for the change review board includes an action such as the following:

```
can(change-review-board,
   check-offerability(change-review-board, CRWS))
```

```
check-offerability(Performer, CRWS)
Effects:        know(Performer, offerable(CRWS)) ∨
                know(Performer, ~offerable(CRWS))
```

I next look at the other actors to which the actor sends messages and determine why messages are sent to those actors. For cases where an actor

spontaneously sends a message, developing the model includes hypothesizing a goal that the actor is attempting to achieve by sending the message. In cases where an actor takes some action in response to a received message, receiving the message is assumed to cause the actor to develop a new goal. For example, by the definition of the request action (see Chapter 2), an actor receiving a request has a goal of performing the requested action.

To make the link between this goal and the observed behaviour of the actor (i.e., the next message sent) is sometimes tricky. It is often helpful to consult the interview data for more information on why a message is sent to a particular actor.

The simplest link is where the action performed is simply the one requested. For example, the customer engineer sends a *Request for classification* message to the project engineer; the project engineer responds by simply returning a *Change classification* message (line 26 of the information-flow model). This can be modelled simply by assuming that the customer engineer performs a request action such as:

```
request(customer-engineer, project-engineer,
 classify-change(project-engineer, change))
```

where the `classify-change` action is defined as:

```
classify-change(Performer, Change)
Effect:        know(Performer, class(Change))
```

and further requests that the project engineer return the result by performing an operation such as:

```
request(customer-engineer, project-engineer,
informref(project-engineer, customer-engineer,
            λX: X=class(Change))
```

The project engineer, on receiving these two requests has the goal of performing the two requested actions; performing them results in the *Change classification* message being returned.

The model for the customer is a bit more complex, involving a bit of indirect reasoning about the effects and preconditions of the actions performed by other actors. Based on the interview data, I model customers as wanting planes which include particular changes. They know that customer engineers can develop change proposals, which the contracts group can implement by building airplanes that include the changes. Therefore, the reason they send change requests to the customer engineer is to get the change proposals back, which they can then have the contracts group implement, in order to get the plane, which is what they actually wanted in the first place.

Finally, it may be simplest to model an actor as simply performing a multi-step process, one step of which is to send a result to another actor. For example, the customer engineer, in developing a change proposal, sends the change request work statement to a variety of groups, including Test Integration, Engineering Cost and Schedules and Weights (line 27 of the information-flow model). Rather than assuming that the customer engineer has a model of what each group does to further the customer engineer's goals, I assume that the customer engineer essentially follows a script for preparing a change request that involves these other groups.

5.2.3 *Example of an intentional model*

Each intentional model consists of four sections. The first three sections list the extralogical components of the models: the sorts (i.e., the types of objects included), predicates and functions.

The sorts form a hierarchy; this is indicated by "isa" relations. For example, the fact that a manager is a kind of person is represented by saying

```
manger isa person.
```

(I have not formally defined the logic of this hierarchy of sorts, but for my representation purposes, this informal use is sufficient.) For each sort I assume a one-place predicate, true of all objects of that sort. For predicates and functions, the sorts of the arguments are shown. Results of functions also have a sort; where this is unclear, it is given explicitly.

The final section presents the knowledge and capabilities of each actor in turn. (In cases where the actor has a long name, an abbreviation for the name is used in presenting the formulas for that actor.) Primitive actions are given in a STRIPS-like notation, as discussed in Chapter 2.

3-1

INTERVIEW OUTLINE

1 Background questions (with key informants)

1.1 The first step in the research plan is to choose a particular process and set of groups to study, with the aid of a key informant, someone who understands the organization.

1.2 structure of organization

1.3 what is the process to be studied

  1.3.1 goal of process

1.4 who is involved (what groups)

1.5 what kinds of information are used

1.6 what kinds of support systems exist

1.7 what media are used

1.8 examples of forms

2 Individual interviews

2.1 background

  2.1.1 name

  2.1.2 job title

  2.1.3 position in organization

  2.1.4 describe your job

2.2 inputs

  2.2.1 what kinds of information do you need to do your job?

  2.2.2 what kinds of information do you receive?

    2.2.2.1 examples

2.2.2.2 example of forms used

2.2.3 how do you get this information?

2.2.3.1 from whom do you get that information?

2.2.3.2 how often do you talk with them?

2.2.4 do you get information you don't need?

2.2.5 is there information you could use that you don't get?

2.3 processing

2.3.1 what do you do with the information you get

2.4 outputs

2.4.1 to whom do you give information?

2.4.2 what kinds of information?

2.4.2.1 examples of messages

2.4.2.2 examples of forms used

2.4.3 how often do you talk with them?

2.4.4 what media do you use? (e.g. phone calls, paper memo)

2.5 support system

2.5.1 how do you use the system

2.5.2 what kinds of information are on the system

3-2

SAMPLE INTERVIEW

Verbatim transcript of part of an interview at Airplanes, Inc. on March 3, 1988 with George, the former manager of Engineer Change Control and Cliff, a former customer engineer who acted as my liason with the company. (Some phrases were unclear on the tape; my best guess at these passages is printed in [square brackets].)

1 George: ...so here we've got a detailed, there you got your detailed spec and
2 the changes that are made when they negotiate with the customer,
3 okay, is called, it's a CR package, a change request package.

4 Kevin: Okay.

5 George: When they discuss the standard aeroplane with the customer, the
6 customer has a wish-list, certain things that he wants for commonality,
7 for logistics, he brings these things in, plus his own people, his
8 handling people, his stewardii [sic] want things in a certain place and
9 so forth, like in the galley, they want the drawers in a certain place,
10 you know what I mean, so he will request a certain type of, will say, a
11 lavatory... with certain features in it, in certain places, and then you
12 gotta go to a lav. vendor to have them make that, okay, now, so, this
13 would come up for a change like that as a CR, part of a CR package.

14 Cliff: CR is change request.

15 George: It's a change request, okay, now, when the team comes back, they have
16 a package, what they call a CR package, and what the engineer does,
17 the customer engineer, he releases the CR package, when, he, he

18 releases the CR package, those items, he releases, by an update to the
19 Itemized Work Statement.

20 Cliff: After the customer's purchased those items.

21 George: Yeah. Yeah. Well, they're run through offerability.

22 Cliff: Yeah. Well, he...

23 Kevin: Okay, now, let me [finish the second point]. So, the CR at some point,
24 someone has to, there's some process...

25 George: Yes.

26 Kevin: ...where the CR... someone goes through the CR...

27 George: Okay. You betcha.

28 Kevin: ...and says, we can do this, we can do this, can't do that one, we can do
29 this.

30 George: What they do, would you say, say he comes back, they [cover] CI, CR
31 package. Okay? The CR package, goes through the Change Board,

32 Kevin: Oh.

33 George: For offerability, it's for offerability, is what it's for.

34 Cliff: This is again before the contract has been signed, so when you go to
35 the customer...

36 Kevin: So it's the same change board that handles MCs?

37 George: Yes. They get the CR package, or no, no, what they do here, when you
38 get the CR package, they're doing working with the basic, it's not
39 change board, it's going through this, document industrial
40 engineering. What happens here is the engineering people get with
41 business management and business management negotiates the CR,
42 the change request, with operations. That negotiation is called, is

43 called, offerability. Offerability, includes two things: a cost and a
44 schedule.

45 Cliff: What doesn't concern George is weight, that's another one of the
46 factors, that they...

47 George: Those are the two things. Okay, now, and what...

48 Kevin: And who does this? The customer engineer and business
49 management?

50 George: And, with the IE people in operations. The IE people, that's the
51 manufacturing side of the house.

52 Kevin: I see, so the IE people are the ones who have all the standard work
53 flows and they look at it and say, oh well, this is a pretty big change,
54 it'll probably take the engineer guys this long to engineer...

55 George: That's where they come up with the A & A, based on when this
56 aeroplane gets delivered, okay, the IE cat looks at this and says, hey,
57 we can do it, or we can't do it, that's acceptance or they reject it, they
58 can't do it. So, but this is for offerability. Now the ones that they can't
59 offer they'll go back and tell 'em, we can't do it, because it takes too
60 long, we can't get it on those aeroplanes. Okay.

61 Kevin: Right, if they came in and said we've decided that we need an all
62 titanium skin, the IE guy would kind of look at that and say, sorry,
63 we're not going to be able to do that one.

64 George: That's correct. Okay, so that CR will go away then. Then the customer
65 would have to agree that he'll take aluminum or fiberglass or
66 whatever's there instead, all right?

67 Kevin: Yeah, because it's conceivable since there's no contract signed at this
68 point that the customer might even come back and say if you can't do
69 that CR, then I'm not interested.

70	Cliff:	Or they might...
71	George:	Might very well.
72 73 74	Cliff:	There might be some horse trading though, telling 'em that you buy the aeroplane and we'll have to commit it as a committed change after it's purchased and run through to try to contract the schedule.
75 76 77 78 79 80	George:	Yeah, well, yeah. That's horse trading is what goes on, when they go to the customer they say, we can not do it basic, but if the customer wants this, it's not by PRR, they'll ask him to write an MC, a Master Change, and [in he goes] the Master Change, the Master Change goes to the board, takes all the fat out of the IE schedule, comes up with a recovery schedule to meet those aeroplanes.
81 82 83	Cliff:	And if you've got the customer engineer telling change board that this is vital for this customer, he wants it, and sales's made commitments and some beyond this aeroplace...
84 85	George:	You get the company position and then you bust your balls; [in plain English] you do everything you can to make it happen.
86 87 88 89 90	Kevin:	So, in other words, ah, okay, so, in other words, it's not really the case that there's this line and everything that's before it goes in as a CR and everything that's after it comes in as an MC. It might be that they look at something that the customer wanted in the contract and say, well, in the ordinary course of events we wouldn't be able to do this...
91	George:	That's correct.
92 93	Kevin:	But since you really want it, we will pretend that you asked us for it late, and, and do... some catchups to get it.
94 95	George:	You have to come in and use a serialized control; you have to request it by MC.
96	Cliff:	It might not be the same price, if it has to go in as an...

97 Kevin: So, so in other words this change process let's me be...

98 George: That's negotiable...

99 Kevin: Yeah, yeah. This change process, it sounds like, is being used not only
100 for, sort of unexpected kinds of changes, but also for things that you've
101 committed to in advance which you just couldn't do any other way.

102 George: You, you can't support it with the basic release schedule. You can't
103 meet it.

104 Cliff: And that would be discouraged as a regular practice, to, [you know]...
105 I'm understanding things along here, correct me if I'm wrong, the, you
106 strive to get a date, a realistic delivery date to the customer before
107 hand so you don't have to go into all these heroics later on to get
108 features on the aeroplane, because as we witnessed this morning, it r-
109 raises havoc with the whole production line if you've got too many
110 special features.

111 Kevin: Oh, I see, so the normal way... of doing that would be that the
112 customer would say I want this and you'd say, fine, we can do that by
113 1992.

114 George: No, what they do what that package is, when we give this to the IE
115 people, this is, is business management side of the house, they'll go in
116 and they'll look at it and they'll say, hey, the earliest aeroplane we can
117 get that package on is line position, uh, 1050.

118 Kevin: Right. We can do it for you...

119 George: So when they go back...

120 Kevin: We can do it for you in 1995.

121 George: Well, no. No, no, no. No. They'll look at it and they'll say 1050. Now
122 maybe, what the feeling is, that the customer would like to have the
123 delivery of this particular aeroplane, let's say, June of 89. Okay? Or,

124 let me see, let's go out ten months. Uh, [garble], November? Let's say
125 he wants October. Of 89. But when we took the CR package for
126 offerability, the IE people come back and say, hey, the best we can do
127 is December. So what we do then is we go back and we tell him his
128 first aeroplane will be delivered in December. So we don't get into this
129 hassle. We can do it with the basic release on the IE schedule and meet
130 the delivery of his first aeroplane.

131 Cliff: If this happened to be a middle east customer and says I want that
132 aeroplane for the hajj, you know, where they transport all the pilgrims
133 to Mecca, there, if I can't have it then, it's not going to be any good to
134 me for a year or so.

135 Kevin: So, so, you, you hold on to for a year.

136 Cliff: Right. Well, what I'm...

137 George: No, what we'll do then is we'll work that MC [big]...

138 Cliff: There'll be some serious negotiating.

139 George: See, what we'll do then is we'll tell 'em, hey, you come back with an
140 MC and we'll get that on your aeroplane. So the MC comes in as a
141 compulsary change and then we can deliver it in November. We
142 compress it to meet, to support the November delivery. That's the
143 exception.

144 Cliff: Good, right. That is the exception. Because if we did that with
145 everybody...

146 George: Hooo... too costly.

147 Kevin: Well, yes, but the story we were hearing today is...

148 George: Too disruptive.

149 Kevin: ...is because of the increased number of leasing planes, that, uh,
150 they're starting to use RRs even more. Uhm. So maybe we can talk

151 about that, in a bit. So we're still talking here about, about, negotiated
152 changes, contract changes, so they will write the CR, maybe some of
153 the CRs will get pushed over and turned into MCs but most of them
154 are going to go into the basic statement of work?

155 George: That's right, the basic statement, SOW, statement of work.

156 Kevin: And the statement of work is really this document that says it's going
157 to be a plane and it's going to have these options and these standard
158 options and these special options which have been negotiated...

159 George: And this configuration.

160 Kevin: ...this configuration, uh, and it ties in with the contract which says it's
161 going to weigh this much and it's going to have these performances,
162 and...

163 Cliff: There's a customer detailed specification generated after the negotiated
164 configuration has been defined. This detailed specification here is the
165 standard aeroplane detailed specification which is never been sold, it's
166 what [company] considers to be the baseline for pricing, weight, etc.
167 And then you, you delete some of the standard variable features in
168 here that are being replaced by the negotiated...

169 Kevin: So then, so then the DIE schedule comes back and says we'll have you
170 that plane, it's line postition 1050.

171 George: Exactly. That's what they do.

172 Kevin: And they shake hands and sign the contract.

3-3

SAMPLE INFORMATION-FLOW MODEL

#	*Sender*	*Message*	*Recipient*	*Actions taken*
1			Customer	determine requirements send *Change request* message to Customer Engineer [2]
2	Customer	*Change request*	Customer Engineer	prepare *Change request work statement* if necessary consult with Project Engineer send *Request for classification* message to Project Engineer [26]
26	Customer Engineer	*Request for classification*	Project Engineer	return *Change classification* message, one of Specification Change, Design Change, Standard Option [27]
27	Project Engineer	*Change classification*	Customer Engineer	send a *Change request work statement* message to Test Integration [4], Engineering Cost and Schedules [6] and Weights [7] if the change was requested before the contract was signed, send a *Change request work statement* message to Change Review Board [3] otherwise, send a *Change request work statement* message to the Manufacturing Change Board [43] and to Engineering Change Control [45]
3	Customer Engineer	*Change request work statement*	Change Review Board	decide if change can be offered in time available send *Offerability* message to Contracts [8]
8	Change review board or Manufacturing change board	*Offerability*	Contracts	wait for *Offerability, Change Request Document* and *Price* messages when all have been received, if change can not be offered then send *Change request rejected* message to Customer [30] otherwise, send *Proposal* to Customer [15]

15	Contracts	*Proposal*	Customer	decide whether or not to accept the change send *Change acceptance* [17] or *Change rejection* [18] message to Contracts
17	Customer	*Change acceptance*	Contracts	add Change Request to contract send *Change acceptance* message to Customer Engineer [19]
19	Contracts	*Change acceptance*	Customer Engineer	add change to Customer Configuration document if contract has been signed then send *Change acceptance* message to Engineering Change Control [44] send revised *Customer Configuration* message to Project Engineer [48]

3-4

SAMPLE INTENTIONAL MODEL

Sorts

```
airplane
change
change-proposal
crws (change request work statement)
person
engineer isa person
manager isa person
```

Functions

```
change(change-proposal)
change(crws)
```

Relations

```
accepted-by(person, crws)
approved-by(person, crws)
have(object, person)
includes-features(airplane, change)
```

Individual models

Customer

```
∃Airplane, Change: want(customer, have(customer,
  Airplane)) ∧ includes-feature(Airplane, Change)
know(customer, can(customer-engineer, develop-
  crws(customer-engineer, Change)))
can(customer, talk-to(customer, customer-engineer))
can(customer, accept-crws(customer, CRWS))
know(customer, can(contracts, implement-crws(contracts,
  CRWS))
```

develop-change-proposal(Performer, Change)

```
Effect:         ∃change-proposal: have(Performer, change-
                proposal) ∧ change(change-proposal) =
                Change
```

accept-crws(Performer, CRWS)

```
Effect:         accepted-by(Performer, CRWS)
```

implement-crws(Performer, CRWS)

```
Preconditions:  accepted-by(customer, CRWS)
Effect:         ∃Airplane: have(Performer, Airplane) ∧
                includes-features(Airplane, change(CRWS))
```

Customer Engineer (ce)

```
know(ce, can(project-engineer, classify-change(project-
  engineer, Change)))
can(ce, lookup-crws(ce, Change))
can(ce, develop-crws(ce, Change))
know(ce, can(customer-engineering-manager, approve-
  crws(customer-engineering-manager, CRWS)))
know(ce, plan(develop-change-proposal(ce, Change),
  sequence(develop-crws(ce, Change), concurrently(
```

```
   inform(ce, test-integration, CRWS),
   inform(ce, engineering-cost-and-schedules, CRWS),
   inform(ce, weights, CRWS),
   inform(ce, change-review-board, CRWS)))))
```

classify-change(Performer, Change)

```
Effect:          know(Performer, class(Change))
```

lookup-crws(Performer, Change)

```
Precondition:  class(Change) = "Standard Option"
Effect:          ∃CRWS: have(Performer, CRWS) ∧
                 change(CRWS) = Change
```

develop-crws(Performer, Change))

```
Effect:          ∃CRWS: have(Performer, CRWS) ∧
                 change(CRWS) = Change
```

approve-crws(Performer, CRWS)

```
Effect:          approved-by(Performer, CRWS)
```

Project engineer (pe)

```
can(pe, classify-change(pe, Change))
```

classify-change(Performer, Change)

```
Effect:          know(Performer, class(Change))
```

4

Computer Systems Co.[1]

In the next three chapter I present the data, collected at the three companies I studied, which served as the raw material for my typology of generic coordination tasks.

In each chapter, I present the details of the study as it was conducted at the particular site and briefly describe the nature of the product produced and the organization of the company. I then present an informal description of the change process. I first describe the general engineering development processes; given that background, I then show how the change processes fit in. Finally, I present the information-flow and intentional models of the engineering change process that I developed for the site.

1 Overview of site

Computer Systems Co.[2] is a division of a large electronics manufacturer. The division produces several lines of minicomputers and workstations and develops system software for these computers. In 1989 the entire corporation had sales of approximately $10 billion and roughly 100,000 employees.

During the year-and-a-half that we were studying the organization, it was reorganized several times. Most of these changes were fairly minor, but a major reorganization took place just as we were finishing our data collection. I will

---

1 The field work and initial data analysis for this site were done by Steve Brobst and me. The models and their interpretation are my work.

2 Pseudonyms have been used for all sites to disguise the identities of the companies studied.

concentrate mostly on describing the processes as they were at the time we studied them, but I will comment on the changes made in the final reorganization.

1.1 Data collection

The model presented in this chapter is based on 16 interviews with 12 different individuals, including 6 software engineers, two managers and three members of support groups and one marketing engineer. The interviews were carried out during 6 trips to the company's engineering headquarters.

In contrast to the other sites in this study, I did not spend much time directly observing the engineering processes at this site because I worked on the analysis with a former member of the software development group who was able to provide some of these details.

It should be noted that data were collected about only one group, the operating system kernel group, because my contact at this company worked in that group. My impression from interviews with a few individuals who worked in different groups is that processes are similar in other software development units, but I have no direct information about them.

1.3 Characteristics of the product

The group I studied was responsible for the development of the kernel of a proprietary operating system, a total of about one million lines of code in a high-level language. The system an end-user would use is composed of modules from several different software development units (e.g., the kernel, network support, compilers, database, etc.). Development of the operating system had started about 5 years before we began our study; at that time, the system had just been initially released.

1.3.1 *General characteristics of software*

Software is different from other products in many ways. One of the most fundamental differences is that software is not a physical product. This has several implications.

First, once the development process is completed, reproduction of the finished product is relatively straightforward, consisting mostly of duplicating tapes and documentation. Therefore, the product is very malleable and almost any change that can be imagined can be made, without the physical constraints of other products, such as the need to change tooling or produce new components. (This is not to imply that changes to software are costless.) As a result, the rate of changes is higher in software than in hardware (A, p. 110)[3].

Second, problems are much more likely to be systematic. If someone finds a problem with a piece of hardware, another item may or may not have the same problem; a problem may be due to a design flaw, but it may also be a random failure. If someone finds a problem with a piece of software, on the other hand, it is likely that every copy of the software has that same problem. This is especially true for system software, which is usually less customized than application software, and for micro or minicomputer software, where the variance in underlying hardware is less. In some sense there is only one product, not multiple instances.

1.3.2 *About operating systems*

The operating system is the basic program that insulates programs running on a computer from details of the hardware. As a result, it is typically quite specific to the details of the hardware. In this case, the operating system software works only with this manufacturer's hardware.

What is an operating system? Traditionally, the operating system's function has been to insulate programmes from the details of input-output devices such as

---

[3] References to the page number of my field notes are given as the source for quotations from subjects.

printers or disk drives. Additional mechanisms allow multiple users to share the processor without interference. Increasingly operating system provides specialized services such as access to a network or database and transaction management.

Structure of the operating system. The system we studied was decomposed hierarchically into several major subsystems, such as the memory manager or file system; each subsystem is further divided into a number of modules. Each module implements a small set of services.

Proprietary operating system. The group I studied developed a proprietary operating system that ran only on this particular vendor's machines. The company also supported a version of UNIX™. For the UNIX system, there is an externally imposed standard that the software has to meet; for the proprietary system, the standard is set internally and can be changed as seem necessary. One interviewee felt that having a fixed standard made the development of UNIX easier, since there was no argument about what the software should or should not do.

1.3.3 *Interdependencies*

Interactions between different parts of the system are not always obvious, since they are not limited to direct physical connections. As a result, the impacts of changes are not always immediately apparent. In principle detecting interdependencies should be fairly straightforward. Different parts of the system depend on each other if one makes use of services provided by the other. For example, in the case of the operating system, the process management system may need to call routines that are part of the file system; therefore, (parts of) the process management code depends on (part of) the file system code. Typically the interface to a module is defined in what is called a "header file." The header file defines any necessary constants, data types and the routines available in the module. The calling module then includes this header file.

™ UNIX is a trademark of AT&T.

Simply looking at these included files overstates the interdependencies, however, since a given module includes many routines all of which are defined in the header file but only a few of which may actually be used. Furthermore, since it is sometimes time consuming to list exactly which other modules a program uses, programmers often use a file that simply defines everything that is likely to be necessary. Overuse of this file masks the real underlying interdependencies by (apparently) making everything depend on everything else.

Interactions can be determined instead directly from the source code of the system, for example, by looking for places where one module calls another. Cross-reference listings can be made that list which modules call which other modules. These listings exist and are used, but they have two limitations. First, the cross-reference does not indicate where modules use data structures from another module (as opposed to calling routines). Second, the cross reference only covers the kernel; it does not show which routines are called by code written by other groups.

As a result of these problems, there seem to be no reliable mechanical means to determine the interactions between different modules. Instead, social mechanisms are used.

Different interfaces are provided for use by different classes of users. Some services are provided for use by programmes written by customers. For example, the file system provides an interface with routines to allow a user's programme to open a file and read data. Other services are provided to other parts of the operating system through internal interfaces. For example, the file system may need the services of the memory manager to allocate space for buffers.

Interfaces are described in several kinds of documents. The behaviour of a piece of code is supposed to match the document describing it, so changing the code requires a revision of the corresponding document. It is the document that is controlled for changes, not the code itself. The change control process attempts

to ensure that all registered users of a document are informed and get a chance to comment when changes are made.

Interfaces provided to the customers are described in the published manuals for system. Other interfaces are provided by one part of the operating system for general use within the system and are called service interfaces. Interfaces provided by one software development unit for the use of another are documented in external specifications. Interfaces used only within a unit may be documented in an internal specification or perhaps not at all. These documents are maintained in a document library. Programmers can request copies of a document and they will be registered as a user of the described interface. At the time of our study, there were 800-900 documents, with about 1000 users total. A total of 15,000 copies of documents had been distributed.

Changes to different kinds of interfaces are managed differently. Customer-used interfaces are subject to change control but essentially never change, since such a revision would require changes to a customers' code making a new release of the operating system incompatible with earlier releases. New functionality is instead be added to the system as a compatible extension.

External specifications are currently not change controlled. The person I spoke to in the change control group said they would like to change control external specifications, since groups outside the software development unit use these interfaces, possibly including groups external to the company. For example, a third party vendor of a system utility may need to access an internal interface to get certain kinds of information.

Anyone internal to the company can use the service interfaces to operating system. These interfaces are less likely to change since many other people use them. Again, changes can often be be made as an addition.

Other interfaces are not described by any official document, since they are only used internally. These interfaces can therefore be changed without going through the change control process. Unpublished interfaces are supposedly used only by a few other programmers. These may change often since the owner and

users of the interface can get together and negotiate any necessary changes. It appears that the developer of such an interface simply remembers who uses it and informs them informally when a change is necessary.

In theory, a programmer should register with the document library if they want to use a documented interface and ask the maintainer if they want to use an undocumented interface. In practice, programmers sometimes borrow a document or copy pieces of someone else's code and therefore do not realize that they are using an internal interface or that they should inform the developer. In some cases, these other programmers are in other groups, so the usual social norms may not apply. These programmers should usually be using a documented interface. Problems arise when changes are made because the owner of the module can not know to inform these people. In fact, the programmers may never know that the interface changed until their code stops working, or, worse, becomes the source of mysterious system errors.

1.3.4 Customer support

The operating system is supported directly by the company. A service division separate from the developers provides a first level of support. Problems with the system that require changes to the code are eventually routed to the developers for action.

Typically customers have a contract with the company for support. Different customers may opt for different levels of support. These levels mostly affect when the customer can get service, e.g., 24 hours a day or only during business hours, and how quickly the service is provided.

1.2 Characteristics of the organization

1.2.1 Structure

Computer Systems Co. is a division of a large manufacturer. The parent company was divided into many divisions, each with a particular set of products. Each division in turn is divided into subdivisions, each responsible for a particular product. For example, in the division I studied, there was a

subdivision for UNIX, one for the proprietary operating system, one for databases, etc. Each subdivision was divided into three units to five units; the proprietary operating system subdivision we studied had units for the low-level kernel, the high-level kernel and design. The division between high and low-level kernel units was described by my contact as somewhat arbitrary, reflecting as much the need to divide the subdivision into two units because of the number of engineers as by any sharp divisions in the product. Each unit in turn was divided into a few sections; a section comprised 3-5 projects; and each project had a manager and 3-10 engineers. Figure 4.1 shows the approximate structure of the software division.

The software development units in this particular subdivision were located together in one building on a large campus. The other software development units whose products comprise the completed system were in other buildings on the campus, in the surrounding area and around world. The proprietary operating systems subdivision had 200-300 programmers in 8 development groups, plus integration and test, peripheral support, etc.

1.2.2 *Communication with other engineers*

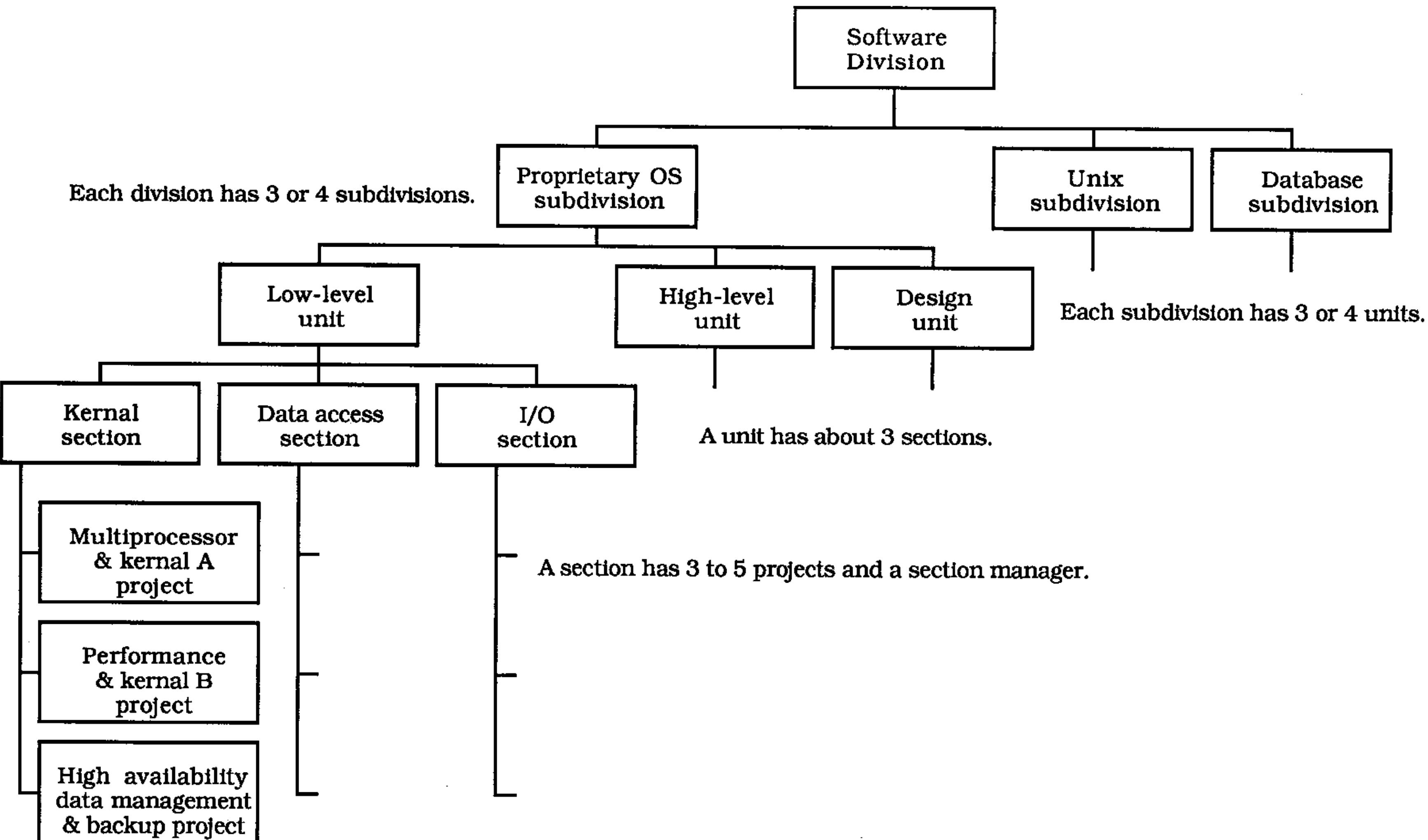
Most communication between engineers seemed to be in face-to-face meetings. The engineers had easy access to electronic mail but they seemed not to use it for discussions between themselves. Face-to-face contact was considered important enough that people from outlying groups were flown in weekly for meetings.

2 The software development process

As with most products, the operating system is continually improved and goes through numerous releases. The goal is to release a new versions of the system periodically, although the releases may not be evenly spaced.

There seem to be two types of release. Major releases get a new version number. Typically, these releases add additional functionality to the system and fix some number of known bugs. Often there is some sort of theme to the release,

Figure 4.1. *Organizational structure of Computer Systems Co. Software division.*



that is, some new class of functionality that is added. Between major releases may be some number of minor releases that fix bugs but probably do not add much functionality. Revisions usually do not affect the existing published functionality (that is, the system has to do everything the previous version did and usually in the same way).

The development of new releases go on in parallel; at the time we visited, 4 or 5 releases were being worked on. The first release (1.2) had been available to customers for a few years and a second (2.0) had just been released. Later releases were in earlier stages of development.

For a new release, process starts with specification of system and especially of new functionality to be added. The designers of the system plan at high level the necessary changes before any code is written and break the system up into pieces so multiple people can work on it. Sometimes the manager looks first at the capabilities of the programmers available and then divides the work accordingly.

For each module, the designers of the system set the specification for what the module does and define the interfaces between modules. One manager commented that these things are much easier to do when you have done it before.

When the functionality is settled, development begins. The development process starts with the code for the previous version of the operating system (which is itself still under development). Under the structure that was in place at the beginning of our study, each software engineer owned a set of modules for all releases. That engineer starts working on a new release by making the necessary changes to the module to add the new functionality. Some modules may not need any changes at all; others may be replaced or new modules created.

Once a module has been coded, the software engineer tests it individually, what is called a unit test. The old functionality of the module can be tested by linking it with the code from the existing release. However, it may be difficult to

test new functionality. Also, a particular change may require new versions of other modules. The engineer can make copies of these modules for the test, but it may be difficult and he may over look some, complicating the testing process. Furthermore, the module may not be tested for interactions with other new modules.

When an engineer finishes making the changes for a new release, he gives copies of the changed modules to the integration group. The integration groups then links the code for different modules to form a complete operating system kernel.

If an error occurs when the integration group tries to compile and link the latest version of the system, they attempt to solve the problem by backing out changes. As a result, submitted changes may sometimes not appear in a released system. This can cause problems, especially if submittals get out of order. First, if an earlier version of a file replaces a later version, the changes made in the later file are lost. Second, a later change may depend on change made in earlier submittals.

The operating system kernel itself can be subject to some tests by the engineers and a testing group in the operating system unit. This is again a unit test, since the entire system includes components from other development groups, such as database. The kernel can be tested by using earlier versions of the code from the other software development units. When bugs are found in the system, the testing group notifies the appropriate engineer by filing a formal bug report.

The finished build is given to another systems integration group to be integrated with the most recent code from other software development units (e.g., networking or database) and the entire system is then tested by the engineers and a testing group. That group does the complete system test and sends bug reports back to the engineers.

Different phases of development have different test requirements. The first level requires that the major functionality be in place, that the system passes

the regression tests indicating that the old functionality still works correctly and that the system runs continuously for 24 hours. The second level test is for 48 hours of reliability and integration with code from other software development units. This level also integrates any fixes made to earlier releases. Once the system passes this level of testing, it is supplied to the other software development units for use in testing their code. The third level calls for 120 hours of reliability, and so on to the sixth and final level, which is the code that can be supplied to customers. There are scheduled dates by which the system is supposed to pass these points, but they may be missed if unexpected problem arise.

As problem are found, the engineers modify and resubmit their modules to the integration group. The operating system kernel can be rebuilt (that is, the changed modules recompiled and the entire kernel relinked) as often as necessary. Depending on the number of changes made, it may take several hours for a build to complete, so there is an upper limit on the frequency of rebuilds. However, if a problem is found immediately and can be corrected quickly, the integration group may make a new build to incorporate the fix on the same day. In order to speed up the process of building the system, the integration group may recompile only those modules that are known to be affected, ignoring some potential interactions.

After the whole system passes a particular level of testing, the integration groups in each of the software development units get all the code for use in testing their next level. Timing problems can arise if other groups are on a different schedule, since they may uncover bugs at different times. If it is important to have the latest code from some other software development unit, for example, to test a change that crosses unit boundaries, the integration people can get whatever is the latest code given to the systems integration people.

Eventually all desired functionality has been incorporated and the entire system successfully passes all its tests. At this point, it may be released to the customer. The exact release date seems to be a function of the schedule and when the system passes the required tests. At that time, all supported customers receive the new release of the operating system. This release includes a

document describing the new functionality and the known problems that have been fixed.

3 The software change process

Now that we have seen the basic design process, we can discuss how changes fit into it.

3.1 Reasons for changes

Lientz and Swanson [,1980 #149] distinguish three reasons for making changes to a programme: corrective, perfective and adaptive.

Corrective changes are those made to fix problems. For this company, problems are defined as disagreements between the behaviour of the programme and the documentation. Many problems are found during the development process by the testing group. Some are found by customers using the product. Fixing these problems usually requires changing the software, but sometimes the fix is made to the documentation instead.

Perfective changes are those made to improve a correct programme with altering its behaviour, typically by improving performance.

Adaptive changes are those that add new functionality, altering the software to meet changing requirements. These changes are mostly made between releases of the system as described above and I will not discuss them much in this section.

3.2 Goals of the change process

The organization had the following stated goals for the change process (A4, p. 7):

> ensure that all critical program parameters are documented: customer commitments, cross-functional dependencies.

ensure that a proposed change is: reviewed by all impacted software development units/functions; formally approved or rejected.

ensure that document status is made available to all users: stable (revision number and date); changes being considered; approved/rejected/withdrawn.

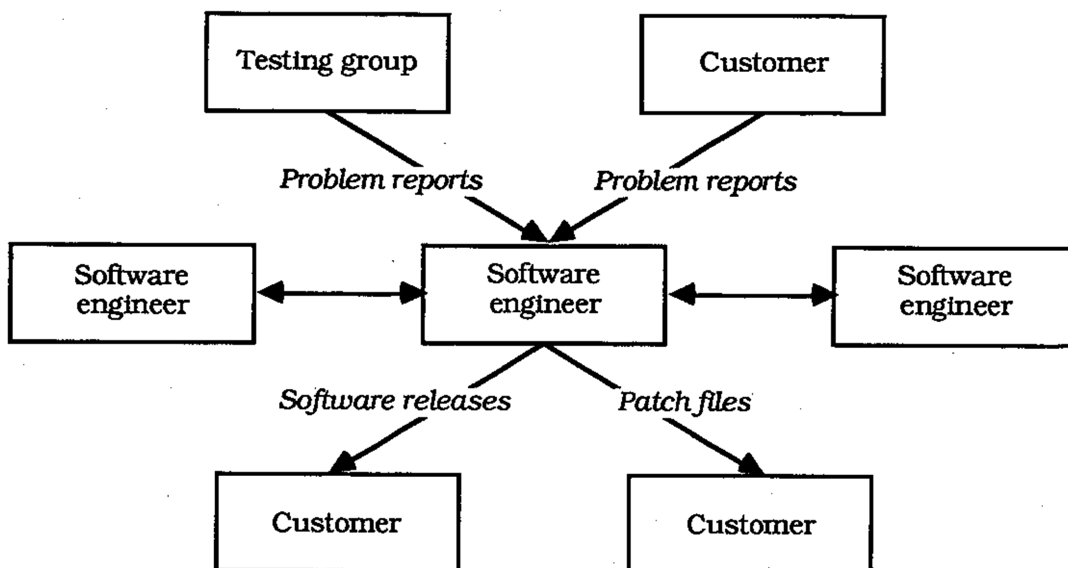
ensure changes are made quickly and efficiently

More generally, I believe the change process actually has two primary goals: to ensure the quality of the software and to minimize the cost of changes. The change system maintains the quality of software by ensuring that changes are made by someone who understands the particular piece of software, that changes are fully tested and that the system and the system documentation are kept in agreement. To reduce the cost of changes, the system requires that that changes be made only in response to a problem or an authorized enhancement. As one manager said (A, p. 108), the "formal change control process is there to prevent changes."

3.3 Overview of process

The basic flow of a change is shown in Figure 4.2. Corrective changes are made in response to problem reports, which come from a testing group and from

Figure 4.2. Overview of change process for Computer Systems Co.



end users of the operating system. These reports are filtered and genuine and novel problems are routed to the software engineer responsible for the apparently affected module. This engineer, in consultation with other engineers, develops a fix for the problem. This fix may require changes to other modules; those changes are made by the engineers responsible for the other modules. Customers get problem fixes either periodically as part of a new release of the software or, for more urgent problems, as a separate patch file that can be loaded on top of the current release.

3.4 An informal description of the change process

3.4.1 Customer service

The software maintenance processes starts when a customer or the testing group notices some problem with a product and complains. The testing group can enter a problem report directly in to the system. Customer complaints are handled in several ways.

Field engineer

One route for problem reports is through the field engineer for the particular site. External customer accounts are managed as a unit, so each customer has a field engineer as a single point of contact with the company. Given a customer problem, the field engineer can create a change request or locate and order patches for the operating system. However, the focus of field engineer's work is changing away from handling problems and more towards doing marketing support and system configuration.

On-line support

A second route for problem reports was just being developed at the conclusion of our study. This was an on-line database of documents. Some documents describe known bugs and give the appropriate workaround or patch information. A customer can dial-in to the database and search for applicable documents by key word to locate a document that describes their problem. If the customer finds a solution to their problem, they can call to order the patch. If

not, they can leave an electronic request for a return call from the response centre.

Response centre

The third and I was told most common route for reporting problems is the response centre. (I was not able to obtain actual counts of the number of reports for each channel.) The company maintains four telephone response centres, with groups of specialists in different aspects of the products. Two centres are located in the US, one on each coast; the others are located in Europe and Asia. Between them, they provide twenty-four hour coverage for the entire world. When a call comes in after hours, it can be routed to an appropriate specialist groups overseas if necessary.

The centre we visited was located near the corporate headquarters. It was the largest of the four and had a total staff of about 300 people in roughly 24 groups. The groups were arranged by product, such as printers or network software. The one group we looked at had 3 hardware specialists and 4 software specialists.

To reach the response centre the customer uses one of several different toll-free numbers. There are different numbers for problems with hardware and software and for ordering supplies, patches, etc.

Initial call screening. An incoming call is answered by a call coordinator. This person first checks the contract status of the caller in a customer database and creates a record of the call in a call database. The call coordinator then attempts to determine which product is causing the problem and records the symptoms of the problem. For software problems, this may include an error number produced by the operating system.

Usually the call is not directly transferred to a specialist; instead, after taking the details, the call coordinator tells customer that someone will call back and ends call. The call might be transferred if the specialist was waiting for that customer to call back, but doing so ties up the toll free number, so they prefer not to.

Electronic requests from the on-line database also enter the system at this point; a call co-ordinator takes the electronic messages and uses the information to create a record in the call database.

Prioritize call. The call coordinator assigns a priority to the call for use by response centre in tracking the call. The usual priority is 4, meaning to call back within 2 hours. Other priorities include 3, meaning to call back within 15 minutes if the customer feels the problem is critical, or 5, meaning to call in the morning if the call comes in after hours and the customer does not have 24 hour support. Priority 2 means the call came from a "down site," meaning one where the system is not working at all; priority 1 is for a call from a field engineer on site who needs assistance.

Route to a specialist. The call is then assigned to the appropriate specialist group, based on the symptoms. The call coordinator may call the group to discuss a problem before making the assignment. Each group has a hot-line for these sorts of calls.

When the call is assigned on the computer, a sheet of paper describing the call prints out in the appropriate group. One of the group members then takes the call. The individual we interviewed worked in one of these specialist groups. She estimated she got six calls a day; on a slow day it might only be 2 or 3; on a very busy day it was as high as 14.

Specialist handles call. If the call appears to be more appropriately handled by a different group, the person handling it may hand carry it to the other group and ask them to take care of it. If they agree, then the call is reassigned on the computer and a new sheet prints out in the new group. This may happen at any point in handling the call.

Based on the initial information in the call record, the call handler may decide to collect various manuals or other information that may be useful. The specialists have many kinds of information available to help solve the problem, including the manuals distributed with the product, any marketing information describing the product and current price lists, ordering information, and

internally produced documents describing the product. Specialists also have access to a a database which contains information about all calls logged by any engineer for the past few years and the database of change requests. In solving the problem, the specialist may make use of flowcharts developed for trouble-shooting problems. In some cases, the call handler may require assistance from someone in another group. For example, a printer may not work because of an underlying problem with the network.

The call handler then calls the customer to discuss the problem further and logs all information on the call record. During this call, the call handler attempts to determine the user's problem.

User misunderstandings. Problem may be due to a user misunderstanding of the product or some question which is answered in the documentation, in which case the call handler explains the problem. Our interviewee estimated that 4 out of the 6 calls per day were things the customer could have looked up in the documentation.

Hardware problems. In some cases there may in fact be a problem with the hardware. Our interviewee estimated that 1/2 to 1 call a day were due to hardware problems. This may be apparent from the beginning of the call; for example, some error codes indicate that a problem is always with the hardware. In other cases, it may take a while to determine the source of the problem. There are hardware specialists in the particular group we studied, but they do not directly deal with the customer; instead, they serve as a resource for the software person and for field engineers. These hardware specialists must sign off on a diagnosis of a hardware problem. In the event of a hardware problem, the call handler can put in a call to a field engineer. The handler can simply turn the problem over to the field engineer or request that the engineer perform some additional tests and then report back.

Software problems. Our interviewee estimated that 1 to 1 1/2 calls a day were real problems with the software. The reported problem may duplicate a known software bug. In this case, if there is a patch, the call handler can order it for the customer. Patch files can be delivered by modem or on a tape. Patches on

tapes come with detailed instructions so customer can install them or have a field engineer install them.

If the problem is still being worked on, there may be no patch yet available. In this case, the call handler can provide any work around given or indicate if the bug will be fixed in a future release. The customer can be given the number of the change request so the customer or the field engineer can track the status of the problem. The response centre does not note that more people have the bug.

The individual we spoke to at the response centre did not get queries from people asking about the status of a change, since customers have other ways to get this information, such as asking their field engineer to follow it for them.

Change request. If the reported problem does not match any existing problem report, the call handler can enter a new change request (a CR). Change requests are stored in a change request tracking database, which assigns it a number. Our interviewee had entered a change request 2 times in 9 months of working in a specialist group.

The change request in the database has sections of text entered by different groups. An initial section describes the problem, the hardware configuration and any known workarounds. Other sections are added by different groups to further explain the problem and give a diagnosis. The response centre would list the field engineer for the site as the reference contact for a change. That way any information or requests for information from the software development units goes back to the field engineer for the site. When a patch is available, the field engineer would be notified and could order the patch for the customer.

The system can generate letters to the customer at different stages indicating the status of the change. The customer gets one when the problem is initially entered. The database is also used to count the number of outstanding requests, to collect and route information, etc. The change request database is the major communication channel between the customer service people and the

development people. One bug, for example, had two CRs against it in the database; the second noted that it was a duplicate of the previous report and asked what was happening to the problem. A development engineer then used the engineering data field to explain the situation.

3.4.2 *Marketing engineer*

Once a change request is entered, it goes first to the marketing engineer for review. There are eight such engineers for the operating system we studied.

Identify problem. The marketing engineers get a list of problems entered on the change request tracking database every morning. About half the time the documentation of the problem is complete enough to work with. In the other cases, the engineer requests additional information from the submitter and waits for it to be sent.

Once the documentation is complete, the marketing engineer attempts to replicate the problem and gain a better understanding of its causes. The engineer may determine that the problem is really a user misunderstanding, a duplicate of a known problem or due to a documentation error. If the problem appears to be genuine, it gets processed further.

Enhancements. The marketing engineer may also decide that the CR is really a request for an enhancement. Sometimes the customer explicitly says it is; occasionally the software unit will reclassify something as an enhancement request. (Sometimes reclassification is used as a strategy to reduce the count of unfixed bugs.) Enhancements are evaluated to determine which are worth doing. Occasionally an enhancement will be given high priority, e.g., if a big customer requests a new feature in order to implement a new system. In some cases, these enhancements may even be issued as a patch, since not every customer needs the new functionality.

Prioritize problem. The marketing engineer then sets the priority for fixing the problem. Low priority problems may never get fixed. One interviewee explained that Computer Systems Co. is sometimes viewed as a "black hole," since reported problems sometimes never get addressed.

Problems that come with a telephone call from response centre or a field engineer get higher priority since someone is waiting for the fix, but if the field engineer does not complain, this does not happen.

If the priority is high, when the problem is finally assigned to a software engineer, the marketing engineer may negotiate informally for a commitment about when a problem is going to be fixed or with the engineer's manager to ensure the engineer has time to work on the fix. If they can figure out a workaround for the bug, then the priority for fixing it may go down. The marketing engineer also decides whether to wait for the fix in a new release or to issue it as a patch.

Assign to software development unit. The marketing engineer next attempts to determine the location of the problem and assigns the change request to the unit responsible for that code. If the marketing engineer can not locate the problem, the change request goes to a SWAT team. This group has access to the system source code. At least, they locate the bug within some module; at the most, they diagnose the bug and suggest a possible fix to the engineer.

The marketing group does not follow a bug once it has been assigned to the software development unit but they do send the followup letter to the customer.

3.4.2 *Engineer develops a change*

Periodically someone in each group in the software development unit checks the database for new change requests for the group. Each request is then assigned to the owner of the particular module.

The software engineer then investigates the problem. If the problem turns out to be entirely in another module, then the engineer passes it to the person responsible for the other module to fix. The engineer first discusses the problem informally with the other engineer; if it is agreed that the problem should be transferred, the engineer then changes the assignment in the change request tracking database.

Once the software engineer has characterized the bug, the status of problem in system is updated to known bug and another letter to the customer is generated at this point saying what the problem is thought to be.

Next the software engineer prioritizes the problem. If the problem is not serious, it may not be corrected for quite sometime (or ever). If the problem is serious, the marketing engineer calls the development group engineer to emphasize that a solution is needed quickly.

Each engineer is simultaneously working on several versions of the system. The problem was probably only reported against one version that the user was using. The engineer needs to check if it also occurs in the other versions. A fix may have to be made separately in each version. For released versions of the operating system, they mostly just fix significant problems; less significant problems might not be addressed in these releases but would be fixed in later releases.

If the problem is internal to a single module, the engineer can just fix the module and resubmit it. In other cases, the change may require changes to several modules or a change in an interface. In these cases, the engineer must negotiate the changes with the owners of the other affected modules as well as other interested engineers.

3.4.3 *Change approval process*

Changes to some interfaces require changes to documents that are controlled by a change control group. To change a document, the engineer writes up a proposal for the change. For a major change, this might be a thick document which is widely circulated. For a minor change, only a few people may be involved. Typically the change includes a marked up copy of the document indicating what has changed.

The engineer investigates who will be affected by the change and invites these people to the design reviews. There are two kinds of reviews: reviews of the external and internal specifications for a module. Other engineers receive the

proposed change. If they are not affected, they simply say so. Otherwise, they come to the design review and comment on design.

When the change is agreed on by the engineers, the engineer seeks approval from the change review board. To make a change, the engineer fills out a change request form. This form indicates the document affected by change, the estimated impact of the change on the schedule, the resources needed, etc.

Changes are assigned a severity level, one of 1) minor corrections such as typographical errors, 2) functional changes to an external specification, or 3) major functional changes that affect many people. The form is then signed by up to 3 levels of supervisors, depending on the severity: the project, section and unit managers. The engineer then submits the form to the change review board. Minor changes do not go through the full change control process; since the change does not affect the functionality of the module, the revised document is simply resubmitted.

3.4.4 *Change review board*

The change review board is the release team, which is composed of representatives of different software development units, software manufacturing, integration, testing and performance unit, 15-20 people in total. This group meets once a week for two hours, during which they discuss change requests (among other issues). For the proprietary operating system, they got 4-5 changes/week; for UNIX, 15/week. This board approves or rejects the submitted change requests.

The change management board is composed of people from program management. The change management board can change the level of a request or escalate the change to higher levels of management.

Before the meeting of the change review board, each member of the team fills out a form evaluating the change from the point of view of the particular group. If there is any disagreement about what to do, the team discusses the changes and comes to some agreement. If necessary, the team members may go back to their groups to get more information about the impact of some change.

Enhancements. If the request is a request for new functionality, it is evaluated and prioritized by the change review board for the product before being implemented. Some changes are suggested directly by customers; requests for new features often come from marketing. In some cases, the change control board decides not to implement the change.

3.4.5 *Engineer changes the module*

Once the change is approved, the engineer changes the code for the modules. The end result is code for a new module that meets the new specification. When the change is complete, the engineer and another engineer walk through the module as a check of the changes. Also, the engineer tests the modules that were affected. To test the module, he recompiles it and relinks the system. For changes that affect multiple modules, the first engineer informally tests the entire change with the other engineers.

3.4.6 *Integration and testing*

When the engineer is satisfied with the change, he submits the new code to the testing and integration group. In order to submit a change, the engineer must note which CR is being addressed; without an CR, the integration group will not accept the change. The submittal form must be approved by the engineer's project manager.

As part of the submittal the engineer notes which build lines are affected and if the files should be carried forward for later builds. If multiple modules are changed, each change is submitted separately. The whole change is assigned a bundle number and each engineer involved submits his changes to that bundle. The initial engineer then indicates when the bundle is complete.

The integration group then recompiles all the latest versions of the code and relinks the modules to make a complete operating system, as described above. The kernel is then tested.

3.4.7 Distribution of changed code

Patches. If the problem is serious, a fix is released in the form of a patch. To make a patch, the object code for the affected modules is released separately on a tape which can be quickly installed on the affected customer's system by a field engineer or by the customer. Patches are maintained by a patch coordinator.

When a new patch is created, the field engineer for the original site is notified, since he was put down as the contact person for the change request. Other people get a software release notice which lists the patch. They can then order the patch from the response centre.

New releases. If the problem did not need an immediate solution (e.g., there is a workaround that avoided the problem), then the customer would wait for a release of the system that included the change. Customers are periodically sent the most recent release of the system.

3.5 Perceived problems with the change process

The process is only partial successful at meeting its goals. There are ways to go around the change control process. For example, in theory changes are made only in response to reported problems. In practice, an engineer can find a bug report that describes a problem in the same module as he wanted to work on anyway and make whatever changes he wants. To some extent, the system and the programmer are working in opposition: the engineer wants to improve the software and the system wants to prevent changes.

Changes are problematic when they are visible outside a single module since they then require coordinated changes to several modules. Changes to the interfaces between modules are always visible and as a result changes to these interfaces are often controlled.

Visible changes pose several problems. For an engineer can only do unit tests (that is, tests of a single module); he or she can not really test the whole operating system since he or she does not necessarily know how the other

modules are supposed to behave. Also, the engineer or even the integration group might not recompile all affected files, since there are no sure-fire methods to determine which other modules might be affected by a change to an interface. Mechanical means of checking for interdependencies exist, but are limited and are made less useful by the overuse of a definition file that essentially makes a module appear to be dependent on all other modules. This may result in modules with inconsistent versions of an interface. These inconsistencies are especially problematic if the modules are developed in different divisions since there is little informal communication between divisions. For example, the word processing system once became the source of mysterious system crashes. It turned out that the developers had used a very low level system call which caused the bug. There was no way for them to know not to use it and no way for the developer of that call to tell they were using it, since the word processor is developed in another, geographically remote software development unit.

If the internal interface is used by an outside company, then there is no informal communication at all, but consequence of changes is greater, since the new release will break customers' code. For example, some system utilities need low-level information about status of jobs in the system; these break when the low-level data changes.

One solution is to provide documented interfaces to the data people want. At the time of our study, the change control group was planning to control the use of these interfaces. They had surveyed customers to see what internal data they use and were planning a new set of interfaces to access that data.

A second solution is to better track what interfaces people are currently using. There is a database that lists which engineers have requested copies of external specification documents can be used to track who uses a particular interface, but it is not clear how often this is used or how accurate it is. For example, engineers frequently borrow copies of documents or code fragments; these borrowings could result in interdependences that are not captured by the database. The change control group was also working on a more formal definition of who should use which interfaces. One person was working on a database that would list all interdependencies, but was concerned that without

formal mechanism for people to report their use of other modules the database would not stay up to date.

3.6 Reorganization

Towards the end of our study, the part of the company we were studying underwent a reorganization. In this section I will concentrate on describing the changes that were made; in a later section I will use the models to suggest an explanation for the change.

Originally, the software engineers in the development group did both development of future release and maintenance of current releases of the operating system, and each software engineer made all changes of both kinds for a set of modules. This arrangement is often called "code ownership," since each module of the system has a single owner. When a bug is reported against a particular module, the owner of the module would check if the bug existed in other releases, determine which releases needed to be fixed, develop a fix for the bug and apply it as necessary.

The reorganization split the development group into support and development groups for the different releases, possibly to allow the development engineers to concentrate their time on adding new functionality. The current releases are no longer under active development; instead, the engineers for these releases just fix important problems. As each version of the system is made available to the customers, development stops and the release goes into support. Many bugs are reported against the current releases, however, since those are the ones that customers have. Therefore, when bug is reported, the support engineers flag that bug may exist in future releases. The development engineers may then fix the problem in the future releases.

Simultaneously, the company also started to use a source control system. This system maintains a copy of all source files. When an engineer wants to make a change to the file, he or she checks it out of the library. The system thus records which engineers are working on a particular module . When the change

has been completed, the module is checked back in and the system identifies what changes were made.

The company did not need such a system under the earlier arrangement because only one person, the owner, would ever modify a source file. In fact, the owner of a module would maintain his or her own copies of the source files. Under the current system, however, the support engineers are not as specialized by module. This seems to be because the support group has fewer engineers because fewer changes are made and no development is going on. The support engineers seem to be organized around change ownership instead of module ownership; that is, an engineer is assigned a particular change and fixes whatever modules are affected. With change ownership, multiple engineers may be working on the same module. In these cases, the engineers may need to manually merge two fixes to ensure that they make sense together.

4 An information-flow model of the change process

The detailed model itself is included as an appendix to this chapter.

4.1 Overview of model

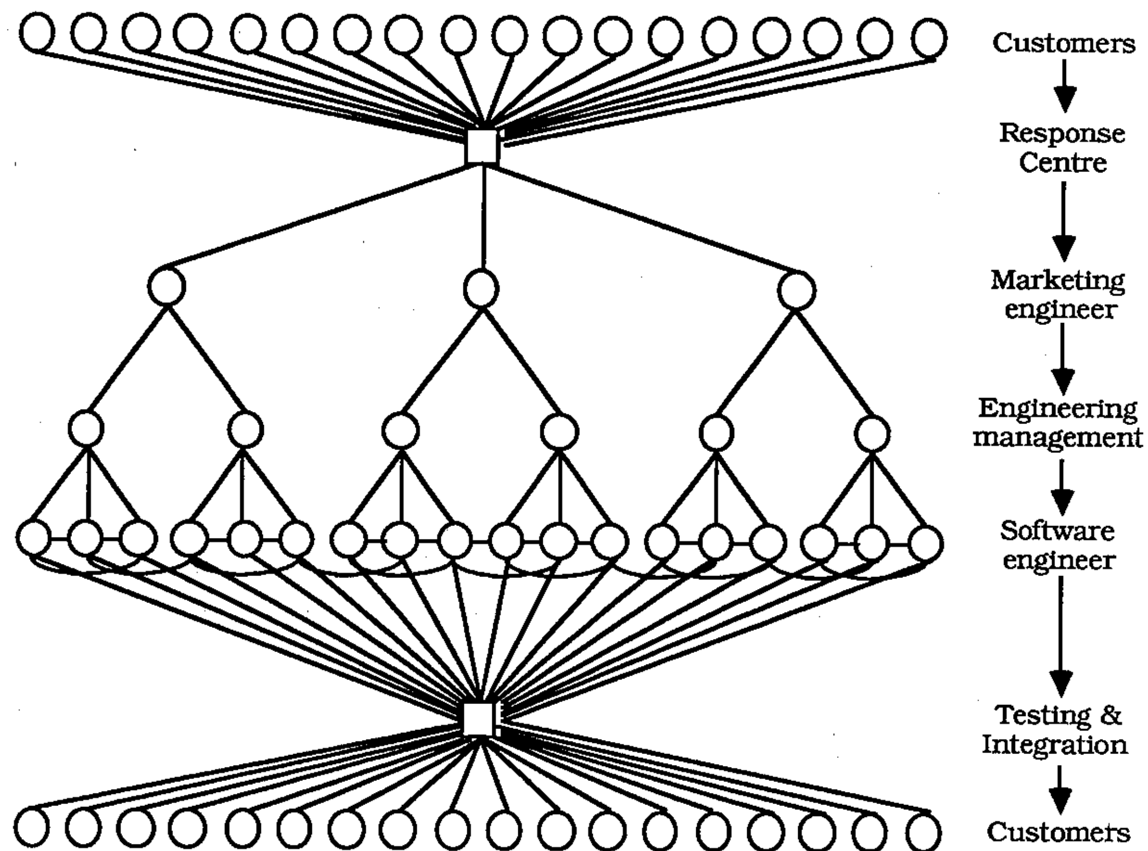
An overview of the information flow model for the software change process is shown in Figure 4.3. Messages flow generally from top to bottom. Only the primary information flow is shown in the diagram; messages do occasionally skip levels or flow backwards to provide feedback, but these links are not shown. Also, only seven of the actors are shown in the overview: customers, the response centre, marketing engineers, engineering group managers, software engineers and testing and integration.

4.2 Components of the model

4.2.1 Actors

The full model has 16 types of actors, described in Table 4.1.

Figure 4.3. Overview of information-flow model for Computer Systems Co.



4.2.2 *Messages*

One interesting feature of this model are the different message types. The different message types and their fields are shown in Table 4.2. There are several different kinds of *Solution* messages: *Clarification, Could not duplicate, Workaround, Software release* and *Patch;* these share some fields and in some cases are processed similarly. Having these all be special cases of a *Solution message* makes the representation of the model more compact.

4.2.3 *Subtasks*

For ease of exposition, I have divided the process into three subtasks, as discussed in Chapter 3: (1) determining that a change is necessary, (2) developing and approving a change and (3) implementing the change.

4.3 Determining a change is necessary

4.3.1 Customer

By assumption, the process starts when the customer encounters unexpected behaviour in the system. The customer sends a message describing the problem to the customer engineer or the response centre. Here we model the customer complaint as a *Problem report* message.

Table 4.1. *Actor types for the Computer System Co. model.*

Type of actor	*Function in change management process*
Bug tracking team	Given symptoms, determine which module needs to be fixed.
Change review board	Approve or disapprove changes to documents under change control; ensure sufficient resources are available, impacts are understood, etc.
Customer	Uses end product; reports problem detected and uses fixes.
Distribution	Distributes new releases of system to customers.
Document library	Store and distributes copies of documents describing system.
Field engineer	Company's representative at the customer site.
Integration	Given copies of source for modules, produces completed system.
Unit manager	Approve proposed changes.
Marketing engineer	Second level of screening of problem reports; sets priority for changes.
Patch coordinator	Maintains copies of patches and fills orders for them.
Project contact	An individual in each project who checks the database for new problem reports and assigns them to the correct engineer.
Project manager	Approve proposed changes.
Response centre	First level of screening of problem reports; attempts to answer questions from database and refers novel problems.
Section manager	Approve proposed changes.
Software engineer	Actually develops or changes code.
Testing	Tests system and reports problems to engineers.

Table 4.2. *Message types for Computer Systems Co. information flow model.*

Message	*Field*	*Explanation*	*Set by*
Problem report	Customer	Customer who had problem	Customer
	Symptoms	Symptoms of the problem	Customer
	Importance to customer		Customer
	Problem number		Response centre
	Site contact	The field engineer for the customer, as a point of contact	Response centre
	Product		Response centre
	Priority		Marketing engineer
	Lab		Marketing engineer or bug tracking team
	Responsible engineer		Engineering group contact
	Diagnosis		Software engineer
Proposed solution	Problem	Description of problem being fixed	
	Approved by change review board		Change review board
	Modules affected		
	Text	Description of the proposed change	

Message	*Field*	*Explanation*	*Set by*
Document change	Problem		
	Document		
	Proposed change		
	Level of change	see text	
	Project manager approval		
	Section manager approval		
	Lab manager approval		
	Change review board approval		
Submittal	Problem	Problem being addressed	
	Reason for change		
	Project manager's approval		Engineering group manager
	Modules affected	which modules are changed, deleted, added, etc.	
	Group with	Other modules that must be submitted	
	Integration instructions	Special instructions about compiling, etc.	
	Depends on	Prior changes that must be included	
	Text	The code for the new module	
Comment	Proposed solution	which proposed solution the comment is about	
	Text	Text of the comment	

Message	*Field*	*Explanation*	*Set by*
Solution	Problem	Problem report for which this is a solution	
	Text	Description of solution or actual code	
Clarification			
Could not duplicate			
Workaround			
Software release	Problems fixed	List of problems fixed in release	
Patch	Patch number	Order number of the patch	
Patch available	Product		
	Problem fixed		
	Patch number	Order number of the patch	
Patch request	Customer	Customer requesting patch	
	Patch number	Order number of the patch	

Since each customer has a single field engineer and there is only one response centre, the customer does not have to do any search to determine the correct recipient of the *Problem report* message.

4.3.2 *Response centre*

The response centre and field engineers do approximately the same things, so here I will only discuss the response centre. When the response centre receives a *Problem report* message it filters out user misunderstandings, hardware problems, etc. For example by looking in the database of previously received *Problem report* messages, the response centre may find that the current problem duplicates a known problem, find a solution to the problem and return some sort of *Solution* message immediately. Otherwise, the response centre picks a marketing engineer based on the product and forwards the *Problem report* message by entering the it in the change request tracking system.

4.3.3 *Marketing engineer*

The marketing engineer also filters out user misunderstandings, etc. The marketing engineer can check if the problem reported is a duplicate of a previously reported problem and if so, immediately return an appropriate *Solution* message. If the problem is actually a request for new functionality, the message is classified as an enhancement request, which so far I don't discuss. Otherwise, he or she prioritizes the problem. Problem reports are assigned in change request tracking database to a particular software engineering project manager, based on the module in which the problem seems to occur.

Determining the module may be quite difficult. In some cases, there is a separate group, the bug tracking team, that specialize in tracking down problems; in other cases, the product engineer may have some default rules that assign the problem to a particular group (who then in essence play the same role).

4.3.4 *Software engineering project manager*

The software project manager forwards the *Problem report* message to the particular engineer who is responsible for the affected module.

4.3.5 *Software engineer*

The engineer may determine that the problem is actually in a different module, in which case he or she can forward the *Problem report* message to the appropriate engineer. Otherwise, the engineer enters the second stage of determining a solution to the problem.

4.4 Solution development and approval

This stage of the process is internal to engineering and involves software engineers, engineering management and change control board.

4.4.1 *Software engineer*

The software engineer gets the *Problem report* message. The engineer first identifies the source of the problem. He or she then prioritizes the problem and puts it in a queue of *problem* messages waiting to be worked on; the priority is set by marketing manager.

Once the problem report has been assigned to a particular software engineer, that engineer must develop a fix that corrects the symptoms of the problem without introducing any new bugs. The engineer develops a possible solution and determines which other engineers will be affected by the change through a variety of mechanisms discussed above. He or she then discusses the proposed solution with various other engineers. This process is modelled by having the engineer send *Proposed solution* messages and wait for the others to return *Comment* messages.

If the *Comment* messages are favourable, then the implementation proceeds; otherwise, the engineer modifies the solution and goes through the discussion process again.

If the change requires changes to a change controlled document, then once the engineer has an acceptable solution, he or she sends a *Document change* message to the appropriate managers to request approval of the change. If the approved *Document change* message is returned, the engineer can submit it to the Change review board for approval. Disapproval of a document change by one of the managers or the Change review board is modelled by having them return a *Comment* message which causes the engineer to revise the proposal.

If the *problem* message was received from another engineer as part of the decomposed problem, then the engineer skips forward to the solution implementation phase, since the proposed solution would already have been prepared, discussed and approved

4.4.2 Other software engineers

When an engineer receives a *Proposed solution* message, he or she evaluates the impact on his module and work schedule and returns a *Comment* message identifying the impact. The engineer may also implicitly agree to the proposed changes to made to his or her modules.

4.5 Solution implementation

The final stage of the change process is the implementation of the proposed solution.

4.5.1 Software engineer

Once the change is approved, the engineer modifies any documents that have changed and sends the new documents to the document centre. He or she then modifies the code for the module and tests the change to assure that it is correct. This testing often is done with another software engineer, but I did not model this process.

Once the module passes its tests, the engineer prepares to submit the change. The submittal must be approved by the project manager; this process is modelled by having the engineer send the *Submittal* message first to the project manager and then sending the approved *Submittal* message to Integration.

The software engineers can only fix their modules of the system. Therefore, if the problem involves someone else's part of the code, they must communicate with that person to fix it. If the change requires modifications to multiple modules, the engineer needs to negotiate the solution with the engineers responsible for the other modules.

I currently model this process by assuming that the *Proposed solution* message contains the details of the decomposition and the negotiation about the change is included in the solution development process above. To implement this change, the first engineer sends a *Problem* messages to the other engineers for each subproblem.

4.5.2 *Integration and testing*

When Integration receives the *Submittal* message, they incorporate the new module into the latest release of the operating system and test the result. If problems are encountered, the engineer is sent a *Problem report* message. Customers are periodically are sent a new version of the operating system by the distribution group.

4.5.3 *Patch coordinator*

The patch coordinator takes new version of a particular module in the *Submittal* message and makes it directly available to the customer. Customers are notified of the patch by a *Patch available* message sent (I believe) to their field engineers. If the field engineer recognizes that the patch fixes the customer's problem he or she sends a *Request for patch* message to the Response centre, who returns a *Patch* message.

5 Intentional model

The model itself is included as an appendix to this chapter.

This model focuses on the coordination around the software engineer, which includes the task assignment process and the process by which engineers look for interactions between modules.

To simplify the model, I did not explicitly include the fact that tasks are assigned to engineers through the STARS database; instead, I assumed that marketing engineers could communicate directly with the software engineers. A second omission is the testing process and error recovery in general. These models show the main flow of the change process; they do not include the (extensive) knowledge necessary to recover when something goes wrong.

4-1

Information-flow model for Computer Systems Co.

#	Sender	Message type	Recipient	Actions taken
7	Marketing engineer	*Problem report*	Bug tracking team	Determine which module needs to be fixed. Forward the *Problem report* message to the appropriate Project contact [8] or Software engineer [10] for the module.
20	Software engineer	*Document change*	Change review board	Approve or disapprove requested change after discussion with all members of the board. If approved, set the approved field and return *Document Change* message [26]. Otherwise return a *Comment* message disapproving the change [11].
1			Customer	Identify a problem with the system. Send *Problem report* message to the Response centre [5] or the Field engineer [18] giving the symptoms and the perceived importance of the problem. Possibly send *Problem report* message to Field engineer [18] to have engineer track problem. Wait for a solution to be announced via the software release notice or by the Field engineer.
16	Marketing engineer	*Could not duplicate*	Customer	If problem still exists, possibly submit more information in another *Problem report* message to the Marketing engineer ***not sure about this*** [6].
4	Response centre, Marketing engineer	*Patch available*	Customer	If the patch fixes an outstanding problem, sends the Response centre a *Patch request* message [15].
3	Distribution	*Software release*	Customer	Install the new software, noting if it solves any outstanding problems.

2	Marketing engineer, Response centre	*Solution*	Customer	Modify the program or install new software and resume work interupted by the problem. If the solution is a workaround, continues to wait for a permanent fix.
19	Integration	*New release*	Distribution	Periodically sends all customers a *Software release* message [3] noting which problems are fixed in the release.
27	Software engineer	*Document change*	Document library	If the change is properly authorized, replace the existing document with the changed document.
22	Patch coordinator	*Patch available*	Field engineer	If the customer still has the problem, send a *Patch request* message to the Response Centre [15].
18	Customer	*Problem report*	Field engineer	Resend the *Problem report* message to Response centre (A2, p. 9) [5]
21	Marketing engineer	*Request for additional information*	Field engineer	Gather requested information and return expanded *Problem report* message [6].
13	Software engineer	*Submittal*	Integration	Check that the submittal has been approved and is in response to a change request. Recompile the module and link it with the other modules into a completed system. If there are problems compiling or linking, back out the change and send the software engineer a *Problem report* message [10]. Test the system. If there are problems with the system, send the software engineer a *Problem report* message [10].

9	Software engineer	*Submittal*	Lab manager	Evaluate the submittal. If appoved, set the approved field and return the Submittal message [23]. Otherwise return a *Comment* message disapproving the change [11].
25	Software engineer	*Document change*	Lab, section or project manager	Evaluate the document change. If approved, set the approved field and return the *Document change* message [24]. Otherwise return a *Comment* message disapproving the change [11].
6	Response centre	*Problem report*	Marketing engineer	If there is not enough information in the report, send a *Request for further information* message to the site contact (usually the Field engineer) [21]. Attempt to recreate the reported problem; if it can not be duplicated, sends the customer a *Could not duplicate* message [16]. Check the database of calls and change requests to see if the problem has already been reported; if it has, then return any existing *Solution* [2] or *Patch available* [4] messages. If the problem reported is actually an enhancement request, then label it as such. Determine the priority of the problem, based on the severity of the problem, its importance to the customer and the availability of a workaround. Determine which module needs to be fixed. If the module can not be determined, forward the *Problem report* message to the Bug tracking team [7]. Otherwise, forward the *Problem report* message to the contact for the software project responsible for that module [8]. If a workaround can be devised, send the customer a *Workaround* message explaining the workaround [2]. Note the work around in the problem database.

14	Software engineer	*Submittal*	Patch coordinator	Add the patch to the list of available patches. Send all field engineers a *Patch available* message [22]. ***not sure about this***
8	Marketing engineer, Software engineer, Bug tracking team	*Problem report*	Project contact	Send the *Problem report* message to the Software engineer responsible for the affected module [10].
15	Customer	*Patch request*	Response centre	Return the requested *Patch* message [2].
5	Customer	*Problem report*	Response centre	Identify the product with the problem. If the problem is a result of a user misunderstanding, then return a *Clarification* message [17]. Check the database of changes to see if the problem has already been reported; if it has, then return any existing *Solution* messages [2]. Otherwise, forward the *Problem report* message to the Marketing engineer for the product in question [6]. If a workaround can be devised, return a *Workaround* message explaining the workaround [2]. Note the work around in the problem database.

11	Software engineer	*Comment*	Software engineer	If the comments are negative, revise the proposed solution and send *Proposed solution* messages as above [12]. If the change requires changing a controlled document, then mark a copy of the document with the proposed change and send a *Document change* message to the lab, section and project managers for approval [23]. Otherwise or when the document change is approved, write the code for the proposed solution. If the change requires changes to other modules, create additional *Problem report* messages requesting those changes and send them to the affected engineers [10]. Test the module by linking it with the most recent release of the other modules on a test system. If any problems are found, revise the code and retest. Once the module passes its tests, send a *Submittal* message to the lab manager for approval [9].
26	Change review board	*Document change*	Software engineer	Send the *Document change* message to the Document library [27]. Continue with the implementation processes.
24	Lab, section or project manager	*Document change*	Software engineer	If more approvals are needed, send the *Document change* message to the next level of management for approval [25]. Otherwise, send the approved *Document change* message to the Change review board for approval [20].

10	Bug tracking team, Project contact, Software engineer	*Problem report*	Software engineer	Determine which module needs to be fixed. If the problem is actually in a different module, then forward the *Problem report* message to the appropriate software engineer [10] or project contact [8]. If a workaround can be devised, send the Field engineer or maybe the response centre a *Workaround* message explaining the workaround [2]. Note the work around in the problem database. If the problem reported is actually an enhancement request, then treat it differently. Characterize the problem and change the status of the problem to known bug. Prioritize the problem and put it on a queue of unsolved problem. When the problem reaches the top of the queue, develop a fix for the problem. If the change was requested by another engineer as part of the solution to another Problem report, then implement the change and send a *Submittal* message to the Integration group [13]. Otherwise, determine which modules and therefore which other engineers will be affected by the change. Send a *Proposed solution* message to these Software engineers [12] and possibly the project manager and wait for comments.
12	Software engineer	*Proposed solution*	Software engineer	Evaluate the proposed solution and returns an appropriate *Comment* message [11].
23	Lab manager	*Submittal*	Software engineer	Send the approved *Submittal* message to the Integration group [13]. If the problem has a high priority, also send a *Submittal* message to the Patch coordinator [14].

17	Testing	Identify some problem with the system. Determine which module needs to be fixed. Send the software engineer [10] or the project contact [8] a *Problem report* message [10].

4-2

INTENTIONAL MODEL FOR COMPUTER SYSTEMS CO.

Sorts

```
engineer isa person
fix
manager isa person
marketing-engineer isa engineer
module
new-release isa fix
person
product
proposed-fix
software-engineer isa engineer
symptoms
```

Predicates

```
approved-by(proposed-fix, manager)
fixes-problem(fix, symptoms)
fixes-problem(proposed-fix, symptoms))
in-product(module, product)
known-problem(symptoms)
responsible-for-module(software-engineer, module)
responsible-for-product(marketing-engineer, product)
uses(module, module)
```

Functions

```
module(symptoms)
product(symptoms)
manager(person)
```

Models of individual actors

Customer

```
know(customer, symptoms)
want(customer, have(customer, Fix) ∧ fixes-problem(Fix,
  Symptoms))
can(customer, talk-to(customer, response-center))
know(customer, can(response-center, determine-fix-for-
  bug(response-center, Symptoms)))
```

determine-fix-for-bug(Performer, Symptoms)

```
Add: ∃Fix: have(Performer, Fix) ∧
     fixes-problem(Fix, Symptoms)
```

Response Centre (rc)

```
can(rc, lookup-fix-for-bug(re, Symptoms))
can(rc, locate-bug-in-product(re, Symptoms))
know(rc, responsible-for-product(Engineer,
  product(Symptoms)) ⇒ can(Engineer,
  determine-fix-for-bug(Engineer, Symptoms)))
marketing-engineer(Engineer) ⇒
  can(rc, talk-to(rc, Engineer))
```

lookup-up-fix-for-bug(Performer, Symptoms)

```
Pre: known-bug(Symptoms)
Add: ∃Fix: have(Performer, Fix) ∧ fixes-problem(Fix,
     Symptoms)
```

```
locate-bug-in-product(Performer, Symptoms)
Add: know(Performer, product(Symptoms))

determine-fix-for-bug(Performer, Symptoms)
Add: ∃Fix: have(Performer, Fix) ∧
     fixes-problem(Fix, Symptoms)
```

Marketing engineer (me)

```
responsible-for-product(me, product(Symptoms)) ⇒
   can(me, lookup-fix-for-bug(me, Symptoms))
responsible-for-product(me, product(Symptoms)) ⇒
   can(me, locate-bug-in-module(me, Symptoms))
in-product(Module, Product) ∧ responsible-for-
   product(me, Product) ⇒ ∃Engineer: know(me,
   responsible-for-module(Engineer, Module))
know(me, responsible-for-module(Engineer,
   module(Symptoms)) ⇒ can(Engineer,
   determine-fix-for-bug(Engineer, Symptoms))
software-engineer(Engineer) ⇒
   can(me, talk-to(me, Engineer))
know(me, can(bug-tracking-team, locate-bug-in-
   module(bug-tracking-team, Symptoms)))
can(me, talk-to(me, bug-tracking-team))

lookup-up-fix-for-bug(Performer, Symptoms)
Pre: known-bug(Symptoms)
Add: ∃Fix: have(Performer, Fix) ∧ fixes-problem(Fix,
     Symptoms)

locate-bug-in-module(Performer, Symptoms)
Add: know(Performer, module(Symptoms))

determine-fix-for-bug(Performer, Symptoms)
Add: ∃Fix: have(Performer, Fix) ∧
     fixes-problem(Fix, Symptoms)
```

Bug tracking team (btt)

```
can(btt, locate-bug-in-module(btt, Symptoms))
can(btt, talk-to(btt, software-engineer))
know(btt, responsible-for-module(Engineer, Module) ∧
   Module = module(Symptoms) ⇒
   can(Engineer, determine-fix-for-bug(Engineer,
   Symptoms)))
```

locate-bug-in-module(Performer, Symptoms)

```
Add: know(Performer, module(Symptoms))
```

determine-fix-for-bug(Performer, Symptoms)

```
Add: ∃Fix: have(Performer, Fix) ∧
     fixes-problem(Fix, Symptoms)
```

Software engineer (se)

```
responsible-for-module(se, module(Symptoms)) ⇒
   can(se, determine-fix-for-bug(se, Symptoms))
responsible-for-module(se, Module) ⇒
   know(se, uses(Other-module, Module)) ∨
   know(se, ~uses(Other-module, Module))
responsible-for-module(se, Module) ⇒
   know(se, uses(Module, Other-module)) ∨
   know(se, ~uses(Module, Other-module))
know(se, responsible-for-module(Other-se,
   module(Symptoms)) ⇒ can(Other-se,
   determine-fix-for-bug(Other-se, Symptoms)))
can(se, propose-fix-for-bug(se, Symptoms))
can(se, implement-fix(se, Proposed-fix))
know(se, equal(implement-fix(se, Proposed-fix),
   integrate-modules(integration, module(Proposed-fix)))
know(se, can(integration, integrate-modules(integration,
   Modules)))
```

```
know(se, plausible(uses(Module2, module(Symptoms)),
  want(Performer, change(Module2)))
```

determine-fix-for-bug(Performer, Symptoms)
```
Add: ∃Fix: have(Performer, Fix) ∧
     fixes-problem(Fix, Symptoms)
```

approve-fix(Performer, Fix)
```
Add: approved-by(Fix, Performer)
```

propose-fix-for-bug(Performer, Symptoms)
```
Add: ∃Proposed-fix: know(Performer, Proposed-fix ∧
     fixes-problem(Proposed-fix, Symptoms))
```

integrate-modules(Performer, Modules)
```
Add: ∃New-release: have(Performer, New-release) ∧
     ∀Module ∈ Modules: fixes-problem(Modules, Symptoms)
     ⇒ fixes-problem(New-releases, Symptoms)
```

implement-fix(Performer, Proposed-fix)
```
Pre: approved-by(Proposed-fix, manager(Performer)) ∧
     approved-by(Proposed-fix,
     manager(manager(Performer)) ∧
     approved-by(Proposed-fix,
     manager(manager(manager(Performer))) ∧
     approved-by(Proposed-fix, change-review-board)
Add: have(customer, Proposed-fix)
```

Integration

```
can(integration, integrate-modules(integration,
  Modules))
fix(Fix) ⇒ know(integration, can(distribution,
  give(distribution, customer, Fix))
```

integrate-modules(Performer, Modules)
```
Add: ∃New-release: have(Performer, New-release) ∧ fixes-
     problem(Modules, Symptoms) ⇒ fixes-problem(New-
     releases, Symptoms)
     for some Modules
```

Distribution

```
fix(Fix) ⇒ can(distribution, give(distribution,
  customer, Fix))
know(distribution, want(Customer, Fix))
  for some Customers
```

Lab, section or project manager

```
can(manager, approve-fix(manager, Fix))
```

approve-fix(Performer, Fix)

```
Add: approved-by(Fix, Performer)
```

Change review board (crb)

```
can(crb, approve-fix(crb, Fix))
```

approve-fix(Performer, Fix)

```
Add: approved-by(Fix, Performer)
```

5

SITE B: CAR CO.

1 Overview of the site

Car Co., a division of an American automobile manufacturer, designs and mass produces five models of automobiles. In 1988, it produced a total of about 300,000 vehicles and had sales of $5-6 billion.

1.1 Data collection

I began my research at this site by discussing the organization and its basic routines with members of the organization and of an affiliated research group. I then made eight trips to the division's engineering facilities and assembly plants for a total of 19 days of interviews and observations. During these visits, I interviewed 29 people in a variety of positions, including individuals who implemented the change control process and the managers of affected groups, both in engineering and in downstream groups.

Most data collection was done in one hour interviews. I also spent approximately one day each with six release or process engineers observing their routines and meetings. The visit day was chosen to include a group meeting. When possible, I collected examples of the paperwork each person worked on and internal documents describing the processes.

In addition, I had access to a study the company had done of its engineering change process, which included an information flow diagram and flowcharts for each individual involved. I also collected documents used to introduce new hires to the engineering groups and processes.

1.2 Characteristics of the product

The automobile industry is central to the American economy. Recently, however, American manufacturers have encountered stiff competition from foreign manufacturers, especially from the Japanese. In general, it takes American firms longer to bring a new car to market and the overall quality of the cars are often lower. Furthermore, many American auto assembly plants are comparatively inefficient.

Assembled automobiles are sold to dealers who in turn sell them to the public, who are in general not particularly knowledgeable about the technology. The market is driven more by styling and image than by technical features, although cost and quality are increasingly important. Therefore, the push to introduce new technologies is mostly internal, driven by a desire to improve quality and reduce cost, rather than by specific demands from the customers.

Automobiles are heavily regulated and the final product must satisfy regulations concerning emissions, safety, etc. To do this requires an audit trail to ensure that the car that is designed is the same as the one that gets built. Once the car is sold, however, its use is not strongly regulated and the company is not responsible for what the end users do with them, although they do track owners for warrantee purposes.

Car Co. designs and manufactures several car models. Some models have similar options and share common systems, while others are unique. Each model changes in varying degrees from year to year. The 1989 and 1990 models of a car, while similar in many respects, will have a slightly different body and some different options and systems and therefore many different parts.

Most car models have numerous options, such as different engines, transmissions, radios, exterior colours, interior furnishings, etc. Providing these options greatly complicates the design and assembly of the automobile. First, each option must be individually engineered and the interactions between different options anticipated. Second, the correct combinations of parts must be ordered, delivered to the plant and made available in the correct order on the

assembly line. Finally, the plant must be able to produce cars with many different combinations of options.

1.2.1 *Parts and suppliers*

Car Co. is primarily an assembler of cars. From their perspective, an automobile is a nothing more than a collection of parts and an assembly process. Deciding to offer a new feature or revise an existing one therefore translates into designing any necessary new parts and revising the assembly process accordingly.

Parts, once designed, are usually not manufactured by Car Co. itself but are instead purchased from suppliers. Some suppliers are also wholly owned subsidiaries of the parent company while others are independent companies. In either case, the production is not under the direct control of Car Co.'s engineers. Instead, Car Co. and the supplier enter into a contractual agreement for a particular set of parts.

The engineer must coordinate his design with these suppliers. For the second two categories of parts, the supplier knows best their capabilities and the engineer needs to consult with them about what they are capable of providing. (In many cases, ideas for improving a part originate with the supplier.) For off-the-shelf parts, there may be a give and take between what the supplier is willing to produce and what the engineer wants. Owned suppliers in particular may be less responsive to an engineer's requests, because they are at least partially insulated from competition or are balancing requests from many divisions of the company and attempting to provide a compromise solution.

Once manufactured, the parts must be tested to ensure they meet the functional specification and quality standards and, in case of detailed control parts, conform to the detailed design. Car Co. must also coordinate with the suppliers to ensure that the parts arrive in time for use in the factory. It may take several months to develop the necessary tooling to mass produce parts, making it difficult to accurately project the availability of a new part.

1.2.2 *Production technology*

The design of an automobile is heavily constrained by the demands of the production process used, which is the assembly line. It is relatively straightforward (albeit expensive) to design and build a single car. To make efficient use of their production technology, however, Car Co. must make thousands of essentially the same car. Car Co. must ensure that the engineering division produces a single design and that sufficient information is available in the design to allow the desired car to be mass produced. In particular, engineering must ensure that all engineers are working with common assumptions and track which parts are currently being used or tested in prototype cars to be able to precisely specify what parts are to be used in the production cars.

Assembly processes must be carefully planned and tested in advance. The steps the assembly process must be decomposed and assigned to the plant workers in units that can be done by a single person in about one minute. Specialized tooling must be developed for many of these operations. At one car a minute, there is little time to recover from any problems and any difficulty is quickly magnified by the number of cars it can affect. Even seemingly insignificant problems may stop the assembly line and keep the plant closed until they are resolved (a so-called "no build" condition), costing the company literally millions of dollars in lost production.

Interactions between engineers and manufacturing personnel in the plant are complicated by the geography. The engineering department is located in the Detroit area, but the plants that actually assemble the cars are located as far away as Texas. Even the closest plant is about fifteen minutes away by car. In these cases, it is difficult for the engineer to personally inspect the production processes.

On the other hand, the constraints of mass production do remove some problems for engineering. First, the inflexibility of the production process itself tightly controls the content of the product. Given a set of parts, the plant can do little more than assemble them (albeit correctly or incorrectly). More

importantly, the cars are not customized for a particular customer. They do have options, but all options are designed in advance.

1.2.3 Commonality

As mentioned above, several of Car Co.'s car models share common features and systems and therefore common parts. Furthermore, these models may be produced simultaneously on the same assembly line, requiring consistent production processes. Commonality reduces Car Co.'s costs by reducing the number of parts that must be re-engineered and the tooling necessary and by allowing larger runs, but it may increase the work necessary to coordinate changes between several uses.

1.3 Characteristics of the organization

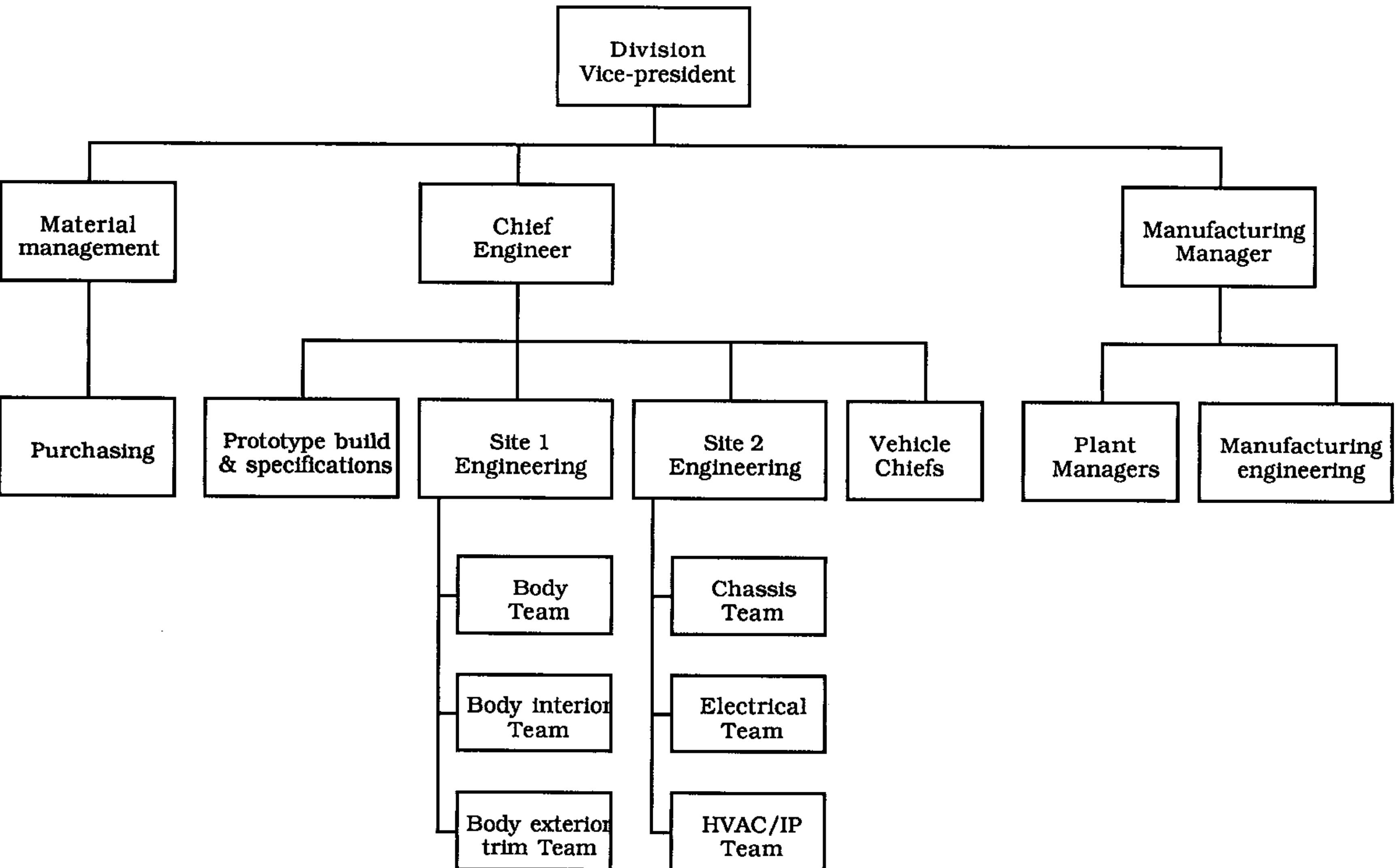
1.3.1 Engineering department

Product development is done by members of the engineering department, headed by the Chief Engineer. Individual engineers are matrixed between functional areas and car models. Figure 5.1 shows the overall organizational structure of the engineering and manufacturing departments.

Work on a particular car model is done under the direction of a project manager, called a Vehicle Chief, who heads a Vehicle Team. The Vehicle Chiefs report (indirectly) to the Chief Engineer. The vehicle teams are the interface between engineering and other groups, such as marketing, who may influence the design of the car.

Physically and organizational, engineers are divided into six functional teams; each team is responsible for a single vehicle system. There are teams for (1) the body, (2) body interior, (3) body exterior trim, (4) chassis and drive train, including engines, (5) electrical, and instrument panel and (6) heating, ventilation and air-conditioning. (For some systems, such as the chassis and drive train, major components such as the engine were developed by engineering groups in other divisions. The system team was responsible for integrating those

Figure 5.1. *Organizational structure of Car Co. Engineering and Manufacturing*



components into the vehicle.) Each team has a head engineer and some number of release engineers, divided into three or four groups, and supporting personnel such as designers and draftsmen.

For historical reasons, the functional teams are geographically divided between two sites with somewhat different procedures (e.g., the sites use different forms with different names for engineering changes and the procedure for approving a change is slightly different). The functional heads report to the head of engineering for that site, who in turn reports to the Chief Engineer.

The functional head engineers have some latitude in their actions and divide responsibilities within their teams somewhat differently. For example, in one team, the head engineer has the VSMT (explained below) consider and approve changes rather than approving them himself. Individual engineers may also interpret their responsibilities differently. As a result, my description will be of the "average" team (or engineer), but any particular team may differ in some details.

Within each team, a single *release engineer* is responsible for a few dozen parts, possibly for several car models. Some engineers work on multiple car years for the same sets of parts; others work only on future or current programs. An engineer may keep working on a particular model year throughout its development or he may turn the design over to another engineer at some point in its development.

Each release engineer has a few *development engineers* working with him. The release engineers determine the necessary functionality and outline of a part; the development engineers then do the actual design. The development engineers report up a different hierarchy to the functional head. Actual detailed drawings of parts are produced by *draftsmen* in a drawing room, which again is supervised in a different hierarchy.

The experimental groups within engineering are responsible for assembling test vehicles to support the early phases of the engineering process. These groups include an experimental specifications group, a small purchasing

department and a garage that assembles prototype cars. A testing group manages the tests that are performed on these vehicles.

The specifications group stands between the engineering division and the other divisions in the company. This group takes an engineer's descriptions of the usage of each part and enters it into a computer system. These formal descriptions are then used by other groups to determine, for example, how many of a given part are necessary to build a given mix of automobiles. Other groups support engineering in a a variety of ways, e.g., by assigning part numbers to new parts.

1.3.2 Material

The material groups handle purchasing parts, scheduling shipments of parts to the plants, etc.

1.3.3 Manufacturing

The assembly process is developed by a separate group of *process engineers*, who report to a head of manufacturing engineering, under the manufacturing manager. These engineers develop tooling for the plant and write production assembly documents which describe how to assemble the parts into a car.

1.3.4 Plant

Each plant has a plant manager and a production manager who also report to the manufacturing manager.

2 The engineering development process

To understand the process of making changes to the design of an automobile, we first must consider how engineering works in general.

The engineering process begins with a high level specification of the desired car, prepared by a marketing group, and produces a design, not an automobile. The design specifies the form and function of each part, describes

how many parts are necessary, how the parts are assembled into assemblies and assemblies into cars and even how the results should be tested.

Work proceeds simultaneously on as many as six model years for each car model. The engineering process for a particular model year begins about five years before the production of the first saleable car and finishes when that model year stops being produced. During this time, the design progresses though numerous design phases.

2.1 Concept development

The earliest stage of design for an entirely new car model is the concept phase, where a new car is developed from the ground up. The timing in this phase is indefinite; many possible cars will be considered but most will never be built. These concept cars will be shown to various groups to gauge their interest in the car; the most promising new designs may be developed further and eventually turned into proposals for a new product. When a proposal is accepted, the development process enters the next phase.

2.2 System definition

The design for a new car or a new model year of an existing car begins with the development of a high-level specification. This phase typically takes about a year and begins about four and a half years before the car begins production (i.e., in the summer of 1990, the 1994 model cars were in this phase).

The outward appearance of the car is designed by a styling group. This group produces a full-scale clay model of the exterior of the car and mockups of the interior, giving the surface definition within which the engineers must work.

Simultaneously, a specification is prepared listing the features and options to be offered or the technology to be used for each car system. Changes between model years are decided by a planning group. For example, the planners might decided that the electrical system of a particular model will be a carry-over (i.e.,

essentially unchanged) in 1990, have small revisions in 1991, be a carry-over in 1992 and be completely redesigned in 1993.

2.3 Design and development

Once the initial high-level description of the car is finished, the car goes into development. This stage also takes about a year, beginning about three and a half years before production (in the summer of 1990, the 1993 model year was in this phase). The final high-level specification should be completed early in this phase.

During this phase, engineers work with suppliers to develop the parts for the car. Each vehicle system and individual part is designed and carefully drafted, usually based on the previous year's design. Between model years there is often a considerable amount (as much as 90%) of "carry-over," meaning that the same parts are used in multiple model years.

To test the functionality of a proposed design, the engineers create mockups or breadboard systems, called preprototype parts [B7, p. 1]. A full-scale model of part of the car, called a buck, may be constructed out of wood. These bucks are used to work out details of the placement of parts or to ensure that there is sufficient space for a new part.

Some tests are performed on the preprototype parts in laboratories on the individual parts or isolated systems; others are done at test tracks with the system installed in a test car. The results of the tests are fed back to the engineers who use them to identify problems with their designs.

During this phase, Car Co. contracts with parts suppliers for samples of any new parts for use in component and systems tests. These suppliers begin supplying samples of the new parts towards the end of the phase (typically about ten copies of each part are produced). These prototype parts satisfy the design, but might not be manufactured by the actual process (e.g., they might be handmade). The parts are used for environmental and functional tests.

To build the sample parts, the supplier must be given the engineer's design. A group called experimental purchasing handles the contracts and payments for prototype parts. This process is carefully controlled because building parts can be quite costly: tooling for a part may cost tens or hundreds of thousands of dollars. Car Co. usually pays for the tooling costs for the sample parts so it owns the tools necessary to build them.

When an engineer finishes the design of a part, he releases it. Releasing a design means making the drawings of parts and assemblies available for use by downstream groups. To release a part, the engineer supplies the the finished drawings and usage information to a specifications writer. These specification writers are physically located in the different engineering groups; each group has between 7 and 15 specification writers.

The specification writer translates the engineer's informal description of the part and enters a formal description in a computer system. For each part, the system stores two kinds of information: a base record, giving the part number and information about the part itself, such as material and finish, and a usage record, saying how many of the part are necessary for each car, where they are used and which engineer is responsible for the part in this use.

The necessary quantity of a part may vary, since some parts are used only for particular options; therefore, each usage may have an option code, indicating the options with which it is used. For example, a standard part used on all cars has no option code; an air conditioner has a code indicating that it is used only on cars with the air conditioner option. If cars with air conditioners need a different part than cars without (e.g., an instrument panel with or without controls for the air conditioner), then one part has a code indicating it is used only on cars with air conditioners and the other, only on cars without.

For prototype cars, the parts information is only needed within engineering and the release is handled by a group called experimental specifications, internal to engineering. When a new part is designed, it is given an experimental part number and the new part information is entered in the experimental parts database. When an engineer orders a prototype car, the

experimental specifications group create a bill of materials, indicating what parts are necessary and orders any necessary stock parts. An experimental build group then builds the vehicles, carefully noting any problems they may encounter.

A prototype car is one that is hand built using the prototype parts from suppliers. Car Co. typically builds on the order of one hundred of these cars for each model each year. Each engineering group initially has a few prototype cars in which to test their systems; some of these prototypes are hybrids, that is, a current production car (e.g., a 1989 car straight off the assembly line) with a particular system upgraded to the model year of interest using pilot (1990) and prototype (1991) parts. Using a production car as the basis for the prototype saves time and money, since much of the car is already built, but it can lead to problems if there are late changes to the production or pilot cars (i.e., to the 1989 or 1990 model year parts). These changes would typically also be intended for the 1991 model year, but they would not appear in the prototype.

Later in the phase, the different systems are integrated in a complete prototype which reflects the current design of the model year. These cars are used, for example, for durability tests or for acceptance rides, where the executives of the division drive the car before approving the design.

2.4 Design validation

The design validation phase takes about eight months and is completed approximately one year before the start of production. The 1992 cars were in this phase during the summer of 1990.

During this phase is the date, called the 100% release date, by which all parts for a particular model are to be released. This year, for example, the 100% release date for the 1991 model of one car was 19 September 1989. Parts are released all through the cycle, though, especially carry-over parts, since there is less uncertainty about their design. Once a part is released, any change requires informing all users of the design and having them redo their work, so engineers typically wait until the last minute to release new parts. In order to allow the

downstream groups time to function, an engineer might prerelease a part, that is, give them copies of the drawings and a preliminary specification, before formally committing to the design.

For a production release, the design is provided by the engineer to the specifications group to be distributed to the various downstream groups in the division for further processing. For a typical car model and year, there were in total about 30,000 releases and rereleases over the five years from concept to introduction [B2, p. 15]. If the part is a new part, it must first be assigned a production part number, which replaces the experimental part number used during development. Next, the base and usage information are entered in the engineering parts database. This information is then transmitted electronically to a central system that checks that the information meets certain technical standards. From there it is redistributed to downstream systems such as the materials management database, which advises the buyers of the need to find a supplier for the new part.

As the designs are finalized, the assembly processes and necessary tools are developed by the manufacturing engineers. The suppliers begin to mass produce the parts and supply samples of production quality parts. These samples, called pilot samples, are built with the actual production tooling using the same processes that will be used for production parts. Pilot samples are used for further tests to ensure the functionality of the design.

2.5 Product validation

During the production validation phase, the basic idea seems to be to test the details of the design and especially the production processes to ensure no bugs are left.

During this phase is a target date for 100% sample approval, that is, the date by which all parts should have been approved by the testing department for use in production cars. These tests are done on the supplied pilot parts and include both tests of material quality and checks that the parts being manufactured actually match the design.

As part of the production validation phase, Car Co. does a pilot build. The 1991 model year cars went through pilot build during early 1990. For pilot build tens of cars are built using pilot parts on the production assembly line, mixed in with the regular production cars. This build is used to test the assembly processes and tooling.

2.6 Production

The final stage of the design cycle is when the car finally goes into production. The 1990 model year started production in early August 1989 and is currently in production. Production continues until mid-summer. Then the plant shuts down for a short time while the production line is reconfigured for the next model year, a process called change-over.

During production, however, an engineer may still be called upon to solve problems in the plant. Most of these problems are found in the first few days of production, called start up. For the first several days, the assembly line is run very slowly, building one car at a time until the various bugs are ironed out. The line speed is gradually increased over the first few weeks until the plant is running at full speed.

2.7 Simultaneous engineering

Historically, engineering designed an automobile and "threw the design over the wall" to the downstream groups to implement. If there were problems with the design, then the downstream groups bounced it back for additional work. This process might be repeated several times if there were additional problems or if the solution did not work. In any event, the downstream groups sometimes had to go to great lengths to work around problems that could have been easily avoided in the design.

To avoid these inefficiencies, the company is now attempting to move to simultaneous engineering. Simultaneous engineering means that the engineers discuss a proposed design with the various downstream groups as they are working on it, to ensure that what they eventually come up with is satisfactory to

all users of design. Other groups can then begin working on their contributions in parallel. Ideally, a design is refined in face-to-face meetings, rather than iteratively. Car Co.'s managers hope that moving these interactions upstream will quicken the design process and result in a better design for an automobile that can be assembled more quickly and with higher quality.

Previously, the various individuals working on the same parts in different groups might never meet directly. As part of simultaneous engineering, various cross-departmental groups have been established. For each vehicle system, there is a group called a Vehicle System Management Team (VSMT) headed by the functional head engineer for that system. The VSMT oversees the design of the entire vehicle system and includes the chief system engineer and the managers from various downstream departments. In one case, the VSMT rather than the functional head engineer gives the final approval for each change proposed.

Each design engineer is a member of a group called a Product Development Improvement Team (PDIT) that helps oversee the parts for which the engineer is responsible. These groups included individuals from departments including purchasing, process engineering and even first-line supervisors from the affected areas of the plant. These individuals can provide the information necessary for the engineer to propose a change. For example, the financial analysis of a proposed change will be done by someone from the financial department, using information supplied by the engineer, purchasing and others. The teams meet regularly to discuss their system or parts; among other things, they track the status of changes that are being made.

3 The engineering change process

In this section, I will describe the engineering change process and show how it fits in the general engineering work flow. I will mostly focus on changes to parts which involve the release engineer. Changes to processes, which do not affect the form, fit or function of parts, will only be briefly touched on. I will first discuss the need for changes and then the processes involved in making these changes.

3.1 Reasons for changes

Parts may be changed for many reasons. From the release engineer's point of view, all changes are changes to the documentation describing the car, either the drawing of the part or the usage information, recording how many of a particular part are used on each car, for each combination of options. The actual parts are supposed to match the documentation, so changing the documentation sets off a corresponding change in the parts. Different kinds of changes result in variations in the change control process.

New model. Most changes are made from one model year to the next. These are often for marketing reasons, so that the next year's car is better than or at least different from this year's. Improvements in the technology of the car are mostly introduced in this way. These changes can be released fairly early in the life cycle of the car and the design frozen.

In principle, the design for a particular year need not change once it has been finalized. In practice, of course, it often does. For a typical car model and year, there were about 30,000 releases in total and 5000 changes over the five years from concept to introduction (B2, p. 15). Changes may be made to both the design of the parts and the processes used to assemble them.

Correcting the documentation. The simplest changes to make are those that do not alter the cars being manufactured, but rather make the design agree with the car. For example, a change may be made to correct an error in the original release, such as releasing too many nuts and bolts. In some cases, a part might not agree exactly with the drawings, but still be acceptable; in this case, the engineer may change the drawings to agree with the actual part, rather than forcing the supplier to change the parts.

Other changes involve changes to both the parts and the documentation. There are two kinds of parts changes, drawing changes and usage changes. If the new part is interchangeable with the old, then the drawing of the part is changed, but the part number and usage stays the same. The part will have the same part number, but a different revision code. If the new part will not be interchange-

able, then it is given a new part number and the use of the old part is cancelled and replaced by the new part. Parts may also be eliminated, for example, if a particular option is dropped. Parts changes may be made for a variety of reasons

Fixing problems. Some changes are made to fix problems with the product that would prevent it from functioning correctly or being correctly assembled in the factory. These changes must be implemented before the car can be mass produced. For example, problems in prototype build may show that a part can not be installed as planned (e.g., it does not fit correctly) and must be redesigned. Other changes are made to correct the production processes. A very few fix safety problems with the car.

Improving the design. Other changes are made to improve an already working product. These improvements are intended to reduce costs, either directly by decreasing cost of a part or indirectly by improving the quality of parts, thus reducing the need for warranty repairs or making the car more saleable. The benefits of these changes can be weighed against the cost of making the changes.

Late changes to the specifications. Changes are often made to satisfy requests from other parts of the company. These requests sometimes take the form of late changes to the high level specification. In theory, the specification should not change once the design process is under way, but the temptation to do so is often great. For example, the marketing department may request a new option or standard feature on the cars to counter an offering by a competitor or higher levels of management may dictate a reduction in the total number of options offered to reduce costs.

Coordinated changes. Finally, a particular interesting set of changes are made to accommodate changes made by other engineers. For example, in order for one engineer to provide a new option, it might be necessary for another engineer to modify existing parts. Alternately, a part used in several models might need to be modified to be used on one model.

Table 5.1. *Rate of changes and cost and number of parts involved in changes at different stages of design.*

Stage	*Rate of change*	*Number of parts*	*Cost of change*
Concept development	High	Only on paper	$1
Design and development (prototype)	High	10	$1,000
Design and product validation (pilot)	Medium	100,000	$100,000
Production	Low	100,000	$1,000,000

3.2 Changes at different phases

The cost and ease of making changes differs greatly in the different phases of the design, as does the need to make changes. (See Table 5.1; the estimates of the costs were provided by my contact at the site).

Concept development through system definition. During the initial phases of the design process, changes are constant as the design is progressively refined. Few aspects of the design are frozen, so anything may change. Since the parts exist only as an engineer's breadboard and no money has been spent on tooling, it costs only about a dollar to make the changes on paper. Therefore there is little need to control changes and no formal change control process.

Design and development. During this phase, parts are released only through the experimental system. The sample parts are relatively expensive (e.g., about $100 each) because they are often hand built, but because they are built in small numbers (typically tens of each part) the total cost is low.

Starting with this phases, changes need to be tracked to know what parts have been used on which prototype cars and how they have been tested. The formal systems necessary are relatively informal, however, since only engineering is involved. In fact, this is probably the major reason for having separate experimental systems.

Design and product validation. Once the parts have been released for production, the change process becomes more formal. Changes require a formal change document approved by engineering management and often must be cost justified.

A large number of changes are made to fix design problems right before the pilot gets frozen. It costs more to make changes at this point, since the production tooling for the parts must be fixed.

Production. Typically there are few changes once the car goes into production,which is fortunate, because changes at this point can cost millions of dollars. A few changes are essential since they resolve "no build" conditions and must be done immediately. Most will be for cost savings, like leaving out an option or an unnecessary step in the assembly process; the source for many are the workers on the plant floor (e.g., foreman calls the engineer and requests a change order [B2, p. 14]) and may be implemented before the change is formally approved.

Increasingly, Car Co. attempts to batch the introduction of changes to give the plant uninterrupted time in which to regain its productivity. Every time a change is introduced, it takes a while for the plant to absorb it and get back to speed. If changes are constantly being introduced, then the plant is constantly struggling to adapt. No changes at all are accepted in the first and last few weeks of the production year. Other changes may be delayed until the next model year, since the changes may involve large numbers of parts that have already been manufactured and could cause a substantial disruption in the plant.

*** cite Kim Clark study showing variance in number of changes was an important negative factor in productivity of plants ***

3.3 Description of the change process

3.3.1 Focus

For this report, I will focus mostly on parts changes made during the latest stages of the change process, after the parts have been released. It is at this point that making changes is most difficult and most expensive and the largest number

of other groups are involved. I will focus initially on the release engineer and on design changes and only touch on process changes. Furthermore, I will describe the current situation, in which simultaneous engineering is being introduced, but is still new.

3.3.2 *Determining a change is needed*

The change process starts with an engineer receiving some report making him decide that a change is necessary. These reports come from many sources, including the plant, suppliers, quality control and marketing.

The marketing department may request specific design changes or features they believe will make the car more saleable, using a formal document called a Car Division Request. Some changes are necessary as a result of a change to the high-level specification of the car.

If the supplier of a part has a question about the design or some problem making the part, their engineers will discuss it with the release engineer, either in person or over the phone. If sample parts fail to fully conform with the design during the approval process, the checking room will inform the engineer so he can determine the necessary fix. As the parts are being tested, the engineer receives reports of the results in a Test Incidence Report.

Once the car is in production, plant personnel may send an engineer a formal report of a problem, using a document called a Problem Report, or may informally request a change by a phone call or by talking to the engineer when he visits the plant. Some problems may be noticed by the engineer himself on a visit.

3.3.3 *Developing a change*

Some changes are straightforward and the engineer simply signs off on the request. For example, if the specifications group points out that more fasteners were released than are necessary, the engineer will probably just sign the necessary change order.

Other requests may require more discussion to determine the nature of the problem. Engineers can discuss problems in their weekly meetings with the PDITs and with the suppliers of the part. They may also visit the plant to see the problem in production cars. Once the nature of the problem is clarified, the engineer begins to develop a solution.

The engineer usually discusses a change with other engineers, based on who he believes will be affected. In many cases, he may just call the engineer and discuss the proposed solution on the phone. The other affected engineers evaluate the proposed change and offer their comments. Other downstream groups may be involved at a PDIT meeting.

In some cases, the solution requires only a change to the assembly process which can be implemented in the plant with the existing parts and tooling. Changes to the assembly process are approved by the manufacturing engineer. The release engineer's approval is only necessary if the part must be modified in some way. In some case, the change is implemented even before it is formally approved. For example, on one visit to the plant, a first line supervisor asked the engineer to change the procedure for attaching a piece of carpeting to allow slicing the carpet. The engineer suggested running 50 cars as an experiment to test the new procedure; the supervisor replied that he had already done that and it worked fine.

If the change requires a new part, the release engineer discusses it with a designer who does the actual development of the new part. As with the initial design, any necessary production drawings are prepared by a draftsman.

3.3.4 Releasing the change

To release a change, the engineer provides the specification writer with a hand written change notice. This change includes a marked up copy of the design showing what is to be changed and any new usage.

The change notice also indicates the desired effective point and what to do with any existing parts. There are four effective points used: on the first saleable vehicle ("SSV"), immediately, obsoleting the stock ("Urgent"), as soon as the

current stock is exhausted ("ASAP") and with the first use of a new option ("WFUO"). SSV is used only when the change is absolutely necessary, e.g., to avoid a safety or emission problem or a no-build condition at the plant. Urgent means to make the change as soon as the parts are available and it is convenient to do so, but until then, cars with the existing parts are acceptable. ASAP indicates that the new and old parts are equally acceptable. WFUO indicates that the new part is a necessary part for a particular option. If a part is used in several model years, there may be different effective points for each year. For example, a change that is required for a 1990 model may be desirable on the 1989 car, and therefore pulled ahead if the parts can be available in time.

For a drawing change, the change will typically be effective and the new parts used after the existing stock of parts is used up ("ASAP"). For a usage change, the process is basically the same, except the choice of an effective point and stock disposition becomes more important since the replacement parts are different from the current stock. The differences may require, for example, that two parts be switched at the same time. These parts may both be included in one change notice or the engineer can indicate that two separate changes must be implemented together.

When the current parts are made obsolete, the engineer can indicate that they should be discarded or reworked to bring them up to date. Parts that are obsolete may be given to service division to be used as spares or just discarded. If there is a large supply of obsolete parts, the materials group may negotiate with the engineer to have them reworked to avoid wasting the parts.

The specification writer prepares a formal engineering change notice that indicates what part is being modified, describes how the part is being modified and gives the reasons for the change. While typing up this change, the specifications writer can check that the information given is correct and that the change is consistent. For example, the specifications writer will check that the part numbers and quantities on the change match those on the drawing. They may check that the new part really will be interchangeable; if it will not be, then the change becomes a usage change instead.

The specification writer then returns the engineer change notice to the engineer for proof-reading and to be formally approved. The marked up drawings are sent to the drafting room to be finished, either by the specifications group or by the engineer himself. Once the drawings are ready, the change package can be issued in the same way as an ordinary release.

3.3.5 *The change is approved and distributed*

To take effect, the change needs to be approved and the engineering change notice signed by the engineer's manager and the functional head engineer for the particular vehicle system. In one of the engineering sites, the change also needs the approval of a change committee. Getting a change approved may require the engineer to submit a financial analysis showing the cost of the change and any expected savings. For some changes, the approval of other groups, such as a certification engineer, may be necessary. Obtaining the necessary signatures usually takes several days, but for important changes the paperwork can be walked through and completed in hours. Once the change notice has been signed, it is returned to the specifications group who distributes it. The new information in the parts database is transmitted electronically to the various downstream groups as for a regular release.

Some changes are complete once they are issued. For example, if an acceptable sample part fails to agree exactly with the design, the engineer may choose to alter the drawing to fit the actual parts rather than force the supplier to alter the parts. In this case, a change notice must be issued to alter the drawings, but the change will be immediately complete. In general, however, the various downstream groups need to act in order to implement the change.

3.3.6 *Other engineering groups prepare for the change*

Once the change document is issued, other engineering groups may official implement their changes. For example, other release engineers may go ahead with other parts of coordinated changes. If the change affects the plant (i.e, part numbers, quantities, options, or illustrations change) then the process engineering group must change the build documents. These documents describe

the steps necessary to assemble and test the car on the plant floor. If necessary, the process engineers may request illustrations of the new parts or assemblies from the design room. At some point, the production tooling must be changed for the new parts. The official notice comes too late in the process to trigger these processes, but it does confirm that the change will take place as agreed.

3.3.7 *Supplier starts to produce new parts*

Once the purchasing group is notified of the change, they pick a supplier for the part and write a new contract to authorize payment for the necessary tooling changes. Usually there is already a supplier, so the choice is clear. Furthermore, the engineer most likely discussed the proposed change with the supplier's engineers, so in many cases, the supplier will already be preparing for the change even before being formally notified. The supplier tells Car Co. the earliest date when the new parts can be ready.

Once the tooling is ready, the supplier starts building sample parts and submits them for approval. These sample parts are checked by the quality control department to ensure that they match the design. If they do, then the parts are approved for use in production. If they do not match, the several things can happen. First, and probably most common, the supplier can fix the part to make them agree. In this case, if the parts are necessary to support production and are usable, the engineer may write a permit to allow them to be used temporarily, even though they are not approved. If the parts are acceptable, the engineer may change the drawings to match the parts.

3.3.8 *The change is scheduled*

A large part of the complication of building cars is deciding what mix of options to build, where to build them and ensuring that the necessary parts will be available to support the production. The scheduling process starts each week when the marketing group gives the schedulers the next 8 week of sales forecasts and three weeks worth of orders for cars.

A group called production scheduling uses these figures to choose how many cars of each model to build, which factories to use, the actual order of the

cars on the line, etc. There may be peculiar constraints that affect this schedule. For example, several car models may be built on the same assembly line and in some cases, cars of a particular model must be separated by a minimum distance on the line to allow enough time for some unique operation. Such a constraint restricts the output of that model to a maximum proportion of the total output.

Based on the production schedule, the material scheduling group determines the parts needed. The production schedule includes the options desired on the cars; this data is used to explode the parts lists to determine the exact numbers of each part needed and when. This material list is then used to create supplier schedules, which tell suppliers how many parts to ship to each plant and when they should arrive.

When material control receives the change notice, they forecast the break point (i.e., when the change will take effect) and determine which plants need the new parts and when. The break point is a function of when the supplier will be able to ship the new part, the stock of the old parts, the plant's schedule and the lead times of any other parts needed for the change. This prediction also must take into account any batching of changes by the plant. In some cases, the supplier may have to produce more of the current part to cover the demand until the new parts can be used.

As the forecast implementation date draws nearer, it is fine tuned, based on actual part availability, amount of remaining stock, etc. A Parts Readiness group oversees the downstream processes to ensure that all necessary parts will be available on time and that nothing falls through the cracks. Three weeks before the forecasted breakpoint, they call to confirm all the data that went into predicting the breakpoint. At this point, material scheduling also shifts to piece control, meaning that they actually count the number of parts left and pick a particular vehicle on which to make the change. Before this time, the implementation date of the change would stay the same, even if number of vehicles being built changed. After this date, it is fixed to a particular vehicle.

3.3.9 Change takes effect

Based on the supplier schedule, the supplier starts to ship the new parts to the plant. The new build instructions are used and the new parts are used on a vehicle. When the change does take effect, the identification number of the first car affected is recorded and the change process is complete.

3.4 Why are changes difficult?

Changes are difficult for several reasons. First, the car is only imperfectly decomposed into independent subproblems. There are many possible interactions between different parts so a change made by one engineer can affect many others. For example, in one car the engine compartment was shaky. To stiffen it, a cross brace was added, but this brace then interfered with an air conditioning pipe.

Second, each engineer operates with incomplete information about the design and the actions of the other engineers. For example, in one car a beam had a hole punched in it, seemingly for no reason. Punching the hole was an additional cost that could be eliminated, except that none of the engineers could remember why the hole was there, making them reluctant to simply omit it. (As it turns out, the hole had been added for a pipe which was later rerouted.)

3.5 Perceived problems with the change process

The managers and engineers at Car Co. perceive a number of problems with their change process (B13, p. 52–59). The first and most common complaint is that there are too many changes. If parts were released once, there would be 13,000 releases for a particular model. In fact, there were 40,000, so each part was released on average three times. (Part of this number is due to carry-overs, since if a part is used in multiple years, then any change must be released separately for each year.) There must be a balance, since the same resources are used to handle changes as the original design.

A second complaint is that the company does not exercise enough discipline in when they introduce changes. One expensive source of changes, for example, are late changes to the high level specification of the car. It is difficult to tell how important a change is and there is a lack of information for making business decisions on when to implement them. In general, the managers want to eliminate bad changes (i.e., ones that have unanticipated negative effects) and move the good ones earlier in the process. One way to do this is to eliminate the need for changes late in the process, by freezing the specification earlier and by releasing and using production parts for the prototype instead of after. Doing this would allow more time to test changes and the final design, hopefully reducing the number of problems.

A final class of problems seem to relate to information that is not fully shared. For example, one member of a downstream group complained about the problems caused when a supplier commits to a change based on discussions with an engineer before the change is fully approved by the various downstream groups. Several interviewees echoed this concern, saying that suppliers should only make changes after receiving approval from supplier management. Usually the limiting factor in making a change is the availability of the new parts, so the supplier begins to make the change based on discussions with the engineer in order to reduce the lead time. Occasionally, however, having the supplier produce new parts too early can be a problem because the old parts are still needed. For example, a change may be delayed because of the plant schedule or problems with other necessary parts or tooling, factors which the downstream groups know but which the individual engineers might not.

Information flow between the engineer and the plant floor can also be disrupted. For example, on one tour of the plant, an engineer noticed that an interior piece for which he was responsible seemed to fit poorly. By walking backwards along the assembly line, he eventually found that the problem was a change that had only been half implemented. Originally the interior piece was held away from the frame of the car with spacers. To eliminate the spacers, the frame piece had been changed to bulge out by the same distance. When the new body pieces were used, the spacers should have been cancelled, but the engineer

found they were still being used, pushing the carpeting out further than intended.

4 An information-flow model of the change process

The information-flow model for this site is presented in an appendix to this chapter.

The model includes a total of 19 kinds of actors; a brief description of each is shown in Table 5.2. Some actors are individuals, e.g., the designer, engineering manager, head of functional area; most are groups, e.g., marketing, drafting room, purchasing.

5 Intentional model of the change process

The full model is included as an appendix to this chapter. Constructing the models revealed several interesting characteristics of the engineering change process in this site.

Formal change process lags actual process. First, the formal change notification process is frequently several months behind the actual implementation of changes. For example, if the change document is sent by mail, getting the authorization signatures can take several weeks or even months. Preparing and distributing the official change notice is also time consuming. As a result, the official notice is usually too late for other release engineers to use it as an initial notification of a change.

Computers support mostly the formal process. Second, computers (and information systems in general) are currently used mostly to support the formal processes. For example, official change notices are entered in and produced by a computer system; in contrast, an advanced engineering change notice might be a photocopy of the change handwrite and a change proposal, a phone call or a face-to-face meeting. In many cases, the actors work around the computer systems to handle advanced information. For example, the parts readiness team maintains a database of new parts, which is created manually from advanced

release information and used to check the development of the on-line official release information.

Table 5.2. Actor types for the Car Co. model.

Type of actor	*Basic function*
Dealers	order and sell cars
Designer	work with layouts to see how parts fit; works out details of design
Drafting room	does actual 3-D drawings with dimensions, as required for production and assembly
Engineering manager	manages a small group of release engineers; approves changes and releases from those engineers
Head of functional area	manages one of seven functional areas; approves changes and releases from engineers in those areas
Marketing	takes orders from dealers and provides sales and sales forecasts to production scheduling determines features of car to be offered
Material scheduling	determine where and when parts are needed and provides suppliers and plants with shipping schedules
Parts readiness	checks that parts will be ready to support production
Plant	assembles supplied parts into finished cars
Process engineer	responsible for development of assembly processes, which are documented in assembly documents
Production scheduling	determines in which factories and what order on line to build ordered cars
Purchasing	manages contracts with suppliers buys samples and get estimates or tooling and lead time estimates

Quality control	check that the parts meet the specifications
Release engineer	responsible for the design of the parts
Service and warranty	feedback from dealers when parts are repaired under warranty or by dealer
Specifications	maintains part database showing engineering intent distributes information about engineering releases to downstream groups
Supplier	manufactures parts
Testing	test prototype cars and notify release engineer of problems that arise
Tooling	develops tools for assembly process

5-1

Information-flow model for Car Co.

#	Sender	Message	Recipient	Actions taken
89			Dealers	forecast sales; order vehicles by sending an *Order* message to Marketing
1	Release engineer	*Change proposal (informal)*	Designer	determine effect of proposed change return a *Change Comment* message possibly treat the *Change proposal* message as a *Change request* and begin preparing for the change, e.g., by working on detailed design
2	Release engineer	***Design work order***	Designer	do layout of new parts or redesigned parts send *Drafting Job Sheet* message to Drafting to have final drawing prepared
85	Drafting room	*Detailed design*	Designer	send engineer a *Detailed design* message
84	Designer	*Drafting Job Sheet*	Drafting room	prepare final drawings return a *Detailed design* message
86	Process engineer	*New illustrationsreq uest*	Drafting room	prepare illustrations of new parts for assembly documents return an *Illustrations* message
3	Release engineer	*Change document (REA or ECS)*	Engineering manager	formally approve or disapprove change; may involve financial analysis of proposed change if approved, marked *Change document* as approved and return otherwise, return a *Change comment* message disapproving the change

4	Release engineer	*Change proposal (informal)*	Engineering manager	determine effect of proposed change return a *Change Comment* message
5	Release engineer	*Change document (REA or ECS)*	Head of functional area	formally approve or disapprove change; may be done by the VSMT; may involve financial analysis of proposed change if approved, marked *Change document* as approved and return otherwise, return a *Change comment* message disapproving the change
6	Release engineer	*Change proposal (informal)*	Head of functional area	determine effect of proposed change return a *Change Comment* message
61			Marketing	determine that a new feature is necessary send a *Change request* message (a ***Car Division request***) to the Release engineer responsible for the appropriate part of the car
62			Marketing	SALES FORECASTING every week, forecast sales for next 8 weeks send *Sales forecast* message to Production scheduling including forecast and orders for next 3 weeks
72	Release engineer	*Change request rejected*	Marketing	possibly resend *Change request* message; possibly drop request
88	Dealers	*Order*	Marketing	use orders for sales forecasting

10	Specifications	*Change notice*	Material scheduling	BREAK POINT CALCULATION set the break point based on lead time for parts, stock on hand and effectivity date from engineer 3 weeks prior to change at plant, call to verify all information used to set the break point pick a particular vehicle on which to implement change
9	Release engineer	*Change proposal (informal)*	Material scheduling	determine effect of proposed change, e.g., possibly negotiate the effectivity, depending on how much the obsoleted parts cost return a *Change Comment* message possibly treat the *Change proposal* message as a *Change request* and begin preparing for the change, e.g., by manually entering parts in the new parts database
8	Purchasing	*Lead time*	Material scheduling	note earliest that new parts can be ready for break point calculation
79	Specifications	*Parts data*	Material scheduling	note parts usage data for material scheduling
7	Production scheduling	*Product schedule*	Material scheduling	MATERIAL SCHEDULING explode parts lists into total number of parts and when they're needed to set supplier schedules send *Supplier schedules* message to suppliers and to plant

12	Release engineer	*Advanced change notice*	Material scheduling {or parts readiness}	put advanced release information into new parts database watch that process is unfolding correctly check that parts are released, contracted, tooling gets done and samples approved if something doesn't happen when expected, send a message to the appropriate group asking about it
63			Plant	notice some condition that affects assembly send *Request for change* message to the Release engineer(or the Process engineer) responsible for the affected parts
14	Process engineering	*Assembly document*	Plant	update assembly documents and retrain operators as necessary use for production process
16	Release engineer	*Change proposal (informal)*	Plant	determine effect of proposed change return a *Change Comment* message possibly treat the *Change proposal* message as a *Change request* and begin preparing for the change
73	Release engineer	*Change request rejected*	Plant	possibly resend *Change request* message; possibly drop request
83	Supplier	*New parts*	Plant	if parts have Sample approval or a Permit, accept new parts use for production process
17	Release engineer	*Permit*	Plant	allow shipments of parts for limited time (e.g, 90 days)

67	Production scheduling	*Production schedule*	Plant	PRODUCTION build cars according to schedule and assembly documents, using parts shipped
15	Quality control	*Sample approval*	Plant	allow shipments of parts for production process
13	Material control	*Supplier schedule*	Plant	allow shipments of parts for production process
18	Release engineer	*Change proposal (informal)*	Process engineer	determine effect of proposed change return a *Change Comment* message possibly treat the *Change proposal* message as a *Change request* and begin preparing for the change, e.g., by making changes to tooling or assembly documents, etc.
87	Drafting room	*Illustrations*	Process engineer	use for assembly document process
19	Specifications	*Advanced change notice* or *Change notice*	Process engineering	ASSEMBLY DOCUMENTS if change affects build instructions (i.e., part number, quantity, option, illustration) prepare new build instructions if new illustartions are needed, send a *New illustrations request* message to the drafting room and wait for an *Illustrations* message send *Assembly documents* message to plant
20	?	*Change*	Production	record first VIN with change
21	Marketing	*Sales forecast*	Production scheduling	create product schedule saying how many cars to build, when and where send *Product schedule* message to Plant and to Material scheduling

23	Specifications	*Change notice*	Purchasing	CONTRACTING if new parts are necessary, pick a supplier send *Contract* message to supplier for new parts
22	Release engineer	*Change proposal (informal)*	Purchasing	determine effect of proposed change return a *Change Comment* message possibly treat the *Change proposal* message as a *Change request* and begin preparing for the change, e.g., by contacting suppliers
69	Specifications	*Detailed design*	Purchasing	send *Detailed Design* message to Supplier
24	Supplier	*Lead time*	Purchasing	send *Lead time* message to Material scheduling
78	Specifications	*Parts data*	Purchasing	use quantities of parts in contracting process
71	Specifications	*Detailed design*	Quality control	use detailed drawings for sample approval process
25	Supplier	*Sample parts*	Quality control	SAMPLE APPROVAL checks that parts agree with drawings if they don't, send a *Sample problem report* message to the Release engineer and Supplier if they do, send *Sample approval* message to the Release engineer, Supplier and Plant

26	Designer	*Change comment (informal)*	Release engineer	if the message points out unanticipated problems, possibly modify the proposal and repeat the process; possibly drop the change if the comments are positive and the change is complete, and release it by sending a *Change handwrite* message to specifications possibly send an *Advanced change notice* to affected groups so they can start preparing in advance of formal notice of approved change
28	Engineering manager	*Change comment (informal)*	Release engineer	if the message points out unanticipated problems, possibly modify the proposal and repeat the process; possibly drop the change if the comments are positive and the change is complete, and release it by sending a *Change handwrite* message to specifications possibly send an *Advanced change notice* to affected groups so they can start preparing in advance of formal notice of approved change
31	Head of functional area	*Change comment (informal)*	Release engineer	if the message points out unanticipated problems, possibly modify the proposal and repeat the process; possibly drop the change if the comments are positive and the change is complete, and release it by sending a *Change handwrite* message to specifications possibly send an *Advanced change notice* to affected groups so they can start preparing in advance of formal notice of approved change

33	Material scheduling	*Change comment (informal)*	Release engineer	if the message points out unanticipated problems, possibly modify the proposal and repeat the process; possibly drop the change if the comments are positive and the change is complete, and release it by sending a *Change handwrite* message to specifications possibly send an *Advanced change notice* to affected groups so they can start preparing in advance of formal notice of approved change
34	Plant	*Change comment (informal)*	Release engineer	if the message points out unanticipated problems, possibly modify the proposal and repeat the process; possibly drop the change if the comments are positive and the change is complete, and release it by sending a *Change handwrite* message to specifications possibly send an *Advanced change notice* to affected groups so they can start preparing in advance of formal notice of approved change
36	Process engineer	*Change comment (informal)*	Release engineer	if the message points out unanticipated problems, possibly modify the proposal and repeat the process; possibly drop the change if the comments are positive and the change is complete, and release it by sending a *Change handwrite* message to specifications possibly send an *Advanced change notice* to affected groups so they can start preparing in advance of formal notice of approved change

37	Purchasing	*Change comment (informal)*	Release engineer	if the message points out unanticipated problems, possibly modify the proposal and repeat the process; possibly drop the change if the comments are positive and the change is complete, and release it by sending a *Change handwrite* message to specifications possibly send an *Advanced change notice* to affected groups so they can start preparing in advance of formal notice of approved change
39	Release engineer	*Change comment (informal)*	Release engineer	if the message points out unanticipated problems, possibly modify the proposal and repeat the process; possibly drop the change if the comments are positive and the change is complete, and release it by sending a *Change handwrite* message to specifications possibly send an *Advanced change notice* to affected groups so they can start preparing in advance of formal notice of approved change
43	Specifications	*Change comment (informal)*	Release engineer	if the message points out unanticipated problems, possibly modify the proposal and repeat the process; possibly drop the change if the comments are positive and the change is complete, and release it by sending a *Change handwrite* message to specifications possibly send an *Advanced change notice* to affected groups so they can start preparing in advance of formal notice of approved change

48	Supplier	*Change comment (informal)*	Release engineer	if the message points out unanticipated problems, possibly modify the proposal and repeat the process; possibly drop the change if the comments are positive and the change is complete, and release it by sending a *Change handwrite* message to specifications possibly send an *Advanced change notice* to affected groups so they can start preparing in advance of formal notice of approved change
29	Engineering manager	*Change document (ER or ECS)*	Release engineer	if approved, send *Change document* message to Head of functional area for approval
30	Head of functional area	*Change document (ER or ECS)*	Release engineer	if approved, send *Change document* message to specifications for distribution
44	Specifications	*Change document (REA or ECS)*	Release engineer	proofread the change document to ensure it agrees with intent send the *Change document* to the engineering manager for formal approval
45	Specifications	*Change notice*	Release engineer	Confirm that change agrees with expectations if not, panic discussions with release engineer? perhaps release associated changes by sending *Change handwrite* message to specifications

40	Release engineer	*Change proposal (informal)*	Release engineer	determine effect of proposed change return a *Change Comment* message possibly treat the *Change proposal* message as a *Change request* and begin preparing for the change, e.g., by contacting suppliers or making changes to other parts
46	Specifications	*Change summary*	Release engineer	check that changes happened as agreed
27	Designer	*Detailed design*	Release engineer	release new design by sending *Detailed design* message to specifications
41	Release engineer	*Request for change*	Release engineer	if requested change does not make sense, return a *Change request rejected* message to requestor of change develop a change to address reported problem, including effective point, etc. if the change is a change to the specifications, not the part, then start release process by sending a *Change handwrite* message to specifications if the change is a parts or usage change, work with affected engineers and downstream groups, by sending them a *Change proposal* message and with the designer by sending a *Design work order* message
47	Specifications	*Request for change*	Release engineer	if requested change does not make sense, return a *Change request rejected* message to requestor of change develop a change to address reported problem, including effective point, etc. start release process by sending a *Change handwrite* message to specifications

49	Supplier	*Request for change*	Release engineer	if requested change does not make sense, return a *Change request rejected* message to requestor of change develop a change to address reported problem, including effective point, etc. if the change is a change to the specifications, not the part, then start release process by sending a *Change handwrite* message to specifications if the change is a parts or usage change, work with affected engineers and downstream groups, by sending them a *Change proposal* message and with the designer by sending a *Design work order* message
35	Plant	*Request for change (a letter)*	Release engineer	if requested change does not make sense, return a *Change request rejected* message to requestor of change develop a change to address reported problem, including effective point, etc. if the change is a change to the specifications, not the part, then start release process by sending a *Change handwrite* message to specifications if the change is a parts or usage change, work with affected engineers and downstream groups, by sending them a *Change proposal* message and with the designer by sending a *Design work order* message

32	Marketing	*Request for change (Car Division Request)*	Release engineer	if requested change does not make sense, return a *Change request rejected* message to requestor of change develop a change to address reported problem, including effective point, etc. if the change is a change to the specifications, not the part, then start release process by sending a *Change handwrite* message to specifications if the change is a parts or usage change, work with affected engineers and downstream groups, by sending them a *Change proposal* message and with the designer by sending a *Design work order* message
50	Testing	*Request for change (Test Incidence Report)*	Release engineer	if requested change does not make sense, return a *Change request rejected* message to requestor of change develop a change to address reported problem, including effective point, etc. if the change is a change to the specifications, not the part, then start release process by sending a *Change handwrite* message to specifications if the change is a parts or usage change, work with affected engineers and downstream groups, by sending them a *Change proposal* message and with the designer by sending a *Design work order* message

42	Service and warranty	*Request for change (Warranty reports)*	Release engineer	if requested change does not make sense, return a *Change request rejected* message to requestor of change develop a change to address reported problem, including effective point, etc. if the change is a change to the specifications, not the part, then start release process by sending a *Change handwrite* message to specifications if the change is a parts or usage change, work with affected engineers and downstream groups, by sending them a *Change proposal* message and with the designer by sending a *Design work order* message
82	Supplier	*Request for permit*	Release engineer	if the parts are acceptable (although not correct) and needed to support the build, send a *Permit* message to the Plant and Supplier, to allow parts to be shipped for a limited time without sample approval
38	Quality control	*Sample problem report*	Release engineer	determine what changes are necesary to the parts if the parts need to be changed, send the supplier a *Request for change* message if the problem is minor, possibly make drawings agree with parts; send a *Design Work order* message to the Designer; send a *Change handwrite* message to Specifications
64			Service and warranty	collect data about cost of repairs done under warranty send *Warranty reports* to the Release engineer responsible for the affected parts
74	Release engineer	*Change request rejected*	Service and warranty	possibly resend *Change request* message; possibly drop request

80			Specifications	notice some error in the specifications, e.g., a part released incorrectly send a *Request for change* message to the Release engineer responsible for the affected parts
51	Release engineer	*Change document (ER or ECS)*	Specifications	if approved, send a *Change notice* (e.g., ***Request for Engineering Action (REA)*** or ***Engineering Change Summary (ECS)***) to other engineers and downstream groups according to distribution list for particular area of car send a *Parts data* message via the parts database to Purchasing and Material scheduling add change to and send *Change summary documents*
52	Release engineer	*Change handwrite*	Specifications	enter change in the parts database and create a formal change document if there seem to be errors in the change notice, return a *Change comment* message to the release engineer pointing out error otherwise, send the release engineer a *Change document* message
53	Release engineer	*Change proposal (informal)*	Specifications	determine effect of proposed change (e.g., is part used in other applications) return a *Change Comment* message
81	Release engineer	*Change request rejected*	Specifications	possibly resend *Change request* message; possibly drop request
68	Release engineer	*Detailed design*	Specifications	once change is released, send *Detailed Design* message to Purchasing, Quality control

65			Supplier	notice some way to improve quality or reduce cost of parts send *Request for change* message to the Release engineer responsible for the affected parts
90	Releas engineer	*Advanced change notice*	Supplier	begin preparing for change begin manufacturing process
58	Release engineer	*Change proposal (informal)*	Supplier	determine effect of proposed change return a *Change Comment* message possibly treat the *Change proposal* message as a *Change request* and begin preparing for the change, e.g., by contacting suppliers, making changes to other parts, to tooling or to assembly documents, etc.
75	Release engineer	*Change request rejected*	Supplier	possibly resend *Change request* message; possibly drop request
55	Purchasing	*Contract*	Supplier	send *Lead time* message to Purchasing stating earliest new parts can be ready begin manufacturing process
70	Purchasing	*Detailed design*	Supplier	MANUFACTURING change tooling for new design manufacture samples of new parts according to design send *Sample parts* message to Quality control build new parts for shipment process
59	Release engineer	*Permit*	Supplier	allow shipments of parts for a limited time (e.g., 90 days) for shipment process

77	Release engineer	*Request for change*	Supplier	change parts to agree with design send *Sample parts* message to Quality control
57	Quality control	*Sample approval*	Supplier	allow shipments of parts for shipment process
56	Quality control	*Sample problem report*	Supplier	negotiate with Release engineer about what needs to change
54	Material control	*Supplier schedule*	Supplier	SHIPMENT if parts have achieved Sample approval or a permit has been issued, ship parts to plant according to schedule if not, and parts are necessary to support build, send a *Request for permit* to Release engineer for affected parts send *New parts* message to Plant
66			Testing	test systems and car if something fails, send a *Test incidence report* to the Release engineer responsible for the affected parts
76	Release engineer	*Change request rejected*	Testing	possibly resend *Change request* message; possibly drop request
60	Specifications	*Change notice*	Tooling	start making new tools

5-2

INTENTIONAL MODEL FOR CAR CO.

Sorts

```
cars
change
change-request
drawing
engineer isa person
feature
layout
manager isa person
part
person
plant
problem
```

Functions

```
part(feature)
part(problem)
part(change)
change(drawing)
change(feature)
change(layout)
change(problem)
manager(engineer)
```

```
manager(manager)
```

Relations

```
approved-by(manager, change)
builds(plant, cars)
have-feature(cars, feature)
have-problem(cars, problem)
responsible-for(engineer, part)
physically-interdependent(part1, part2)
```

Individual models

Marketing

```
can(marketing, determine-feature(marketing))
know(marketing, responsible-for(Engineer, part(Feature))
   ⇒ can(Engineer, implement-feature(Engineer,
   Feature)))
know(marketing, responsible-for(Engineer, Part)) (for
   some Engineers and Parts)
can(marketing, talk-to(marketing, Engineers)) (for some
   Engineers)
```

determine-feature(Performer)

```
Add: ∃Feature: know(Performer, Feature) ∧
     want(Performer, builds(plant, Cars) ⇒ have-
     feature(Cars, Feature))
```

implement-feature(Performer, Feature)

```
Add: builds(plant, Cars) ⇒ have-feature(Cars, Feature)
```

Testing

```
can(testing, find-problem(testing))
know(testing, responsible-for(Engineer, part(Problem))
   ⇒ can(Engineer, fix-problem(Engineer, Problem)))
```

know(testing, responsible-for(Engineer, Part)) *(for some Engineers and Parts)*

can(testing, talk-to(testing, Engineers)) *(for some Engineers)*

find-problem(Perfomer)

Add: ∃Problem: know(Performer, Problem) ∧ want(Performer, builds(plant, Cars) ⇒ ~have-problem(Cars, Problem))

fix-problem(Performer, Problem)

Add: builds(plant, Cars) ⇒ ~have-problem(Cars, Problem)

Release engineer

know(re, responsible-for(Engineer, Part)) *(for some Engineers and Parts)*

can(re, talk-to(re, Engineers)) *(for some Engineers)*

know(re, physically-interdependent(Part$_1$, Part$_2$)) *(for some Parts)*

can(re, evaluate-interdependencies(re, Part))

know(re, plausible(∃Part$_2$: want(re, Change) ∧ physically-interdependent(part(Change), Part$_2$), ∃Change$_2$: want(re, Change$_2$) ∧ part(Change$_2$) = Part$_2$))

know(re, can(designer, develop-detailed-change(designer, Change)))

know(re, can(drafting, develop-drawing(drafting, Detailed-change)))

know(re, can(manager(re), approve-change(manager(re), Change)))

know(re, can(specifications, release-drawing(specifications, Drawing)))

know(re, change(Drawing) = Change ⇒ equal(implement-change(re, Change), release-drawing(specifications, Drawing))

```
know(re, responsible-for(Engineer, part(Change)) ⇒
   can(Engineer, develop-change(Engineer, Change)))
know(re, equal(implement-feature(re, Feature),
   develop-change(re, change(Feature)))
know(re, equal(fix-problem(re, Problem),
   develop-change(re, change(Problem)))
```

approve-change(Performer, Change)
Add: approved-by(Performer, Change)

develop-change(Performer, Change-request)
Add: ∃Change: know(Performer, Change) ∧ Change =
     change(Change-request)

develop-detailed-change(Performer, Change)
Add: ∃Detailed-change: know(Performer, Detailed-change)
     ∧ change(Detailed-change) = Change

develop-drawing(Performer, Detailed-change)
Add: ∃Drawing: know(Performer, Drawing) ∧
     change(Drawing) = change(Detailed-change)

evaluate-interdependencies(Performer, Part)
Add: know(Performer, physically-interdependent(Part,
     $Part_2$)) *for some* $Part_2$*'s*

implement-change(Performer, Change)
Add: builds(plant, Cars) ⇒ has-change(Change, Cars)

release-drawing(Performer, Drawing)
Pre: responsible-for(Engineer, part(change(Drawing))) ⇒
     approved-by(manager(Engineer), change(Drawing))
Add: have(downstream-groups, Drawing)

Engineering manager

can(manager, approve-change(manager, Change))

approve-change(Performer, Change)
Add: approved-by(Performer, Change)

Head of functional area

```
can(manager, approve-change(manager, Change))
```

```
approve-change(Performer, Change)
Add: approved-by(Performer, Change)
```

Designer

```
can(designer, prepare-layout(designer, Change))
```

```
prepare-layout(Performer, Change)
Add: ∃Layout: have(Performer, Layout) ∧ change(Layout) =
     Change
```

Drafting room

```
can(drafting-room, prepare-drawing(drafting-room,
  Layout))
```

```
prepare-drawing(Performer, Layout)
Add: ∃Drawing: have(Performer, Drawing) ∧
     change(Drawing) = change(Layout)
```

Specifications

```
know(specifications,
  equal(implement-change(specifications, Change),
  concurrent(inform(specifications, tooling, Change),
  inform(specifications, purchasing, Change),
  inform(specifications, quality-control, Change),
  inform(specifications, material-scheduling, Change),
  inform(specifications, process-engineering, Change),
  inform(specifications, release-engineer, Change))))
```

6

SITE C: AIRPLANES, INC.

1 Overview of site

Airplanes, Inc. is a manufacturer of commercial jet aircraft. It is a division of a corporation that had total sales in 1988 of $15-20 billion and roughly 150,000 employees.

1.1 Data collection

I began my research at this site by discussing the organization and its engineering processes with members of a central engineering support group. I visited the company's engineering and manufacturing facilities on three occasions, for a total of 13 days of interviews and observation. During these visits, I interviewed a total of 20 people, including managers, engineers and members of various support groups. I also attended several meetings of various change control groups.

Most data collection was done in one hour or longer semi-structured interviews. I also spent an afternoon each with an engineering manager and an engineer observing their daily routines. One such visit included a group meeting. When possible, I collected examples of the paperwork each person worked on and internal documents describing the processes. I also attended meetings of two change control boards.

Finally, I attended a one day class that provided an introduction to engineering operations for new employees. I also collected copies of various

training materials that extensively document the engineering and engineering change processes, totalling several hundred pages of flowcharts and descriptions.

1.2 Characteristics of the product

This discussion of the aerospace industry draws on ("Civil aerospace," 1988; Barker, 1989; Covert, 1989; Simpson, et al., 1988) and especially (Newhouse, 1983).

Technically, Airplanes, Inc. is in the airframe industry; other companies design and manufacture the engines that power the aircraft. The airframe industry has total sales of $50 billion a year and is one of the largest American exporters. In addition, it is one of the few industries that American firms still dominate, although in recent years Airbus has grown considerably. In addition, Japanese companies are known to be interested in entering the industry ("Sincerest form," 1989); they already make large portions of the Boeing 767 among others ("Forget," 1988).

1.2.1 Complex product

Commercial jets are among the most complex of products. A 747, for example, has 175 miles of wire and literally millions of parts ("Trying times," 1989), purchased from 2000 suppliers ("Strain," 1989). Furthermore, aircraft make extensive use high technology and new materials. Reducing the weight of an aircraft by even one pound saves a large amount of fuel—and therefore money—over the life of the plane.

Long product life cycle. One interviewee estimated that it takes about 3 1/2 years from the start of development of a new aircraft to rolling out the first plane and as long as a year after that for testing and certification. Developing a new jet engine takes even longer, from 4 1/2 to 5 years. The lead time for certain parts is also long; for example, landing gear struts may take more than two years to make. These parts must therefore be ordered even before the final design is completed. A particular model of airplane may stay in production for as long as twenty years, undergoing constant evolutionary changes.

An individual jet may be used for as long as twenty years. During this time, of course, many of the components of the plane will be rebuilt or replaced. The lifetime of an airframe itself is often measured in takeoff-landing cycles, since it is the pressurization and depressurization of the body of the jet that most stresses the structure.

Learning curve. Manufacturing aircraft has a steep learning curve: the first planes built are much more expensive than later ones. The combination makes pricing aircraft quite tricky. The price must be set based on an average cost that may not be achieved until the 400th airplane, which might not be delivered for years; as a result, a program make take 10 year or longer to become profitable. The initial sales of a new airplane model are particular important, since they serve to drive the process the furthest along the learning curve.

If the plane does not sell well, the program may never become profitable. Lockheed, for example, left the commercial jet aircraft industry in 1981, after loosing an estimated $2.5 billion on the development of the L-1011. At this point, no single company has the resources to develop an entirely new aircraft alone.

1.2.2 *Regulatory environment*

Design standards. Building airplanes is an exacting business. To ensure the safety of the airplanes, the Federal Aviation Administration (FAA) sets technical standards for the design of the aircraft. These standards are enforced with a unique self-regulating system in which much of the oversight authority of the FAA is delegated to engineers of the aircraft company itself. These engineers, known as Designated Engineering Representatives (DERs), review the design and design processes to ensure compliance with Federal Airworthiness Regulations (FARs).

This delegation of authority is necessary for two reasons: first, the FAA does not have sufficient manpower to check the entire design; and, second, because engineers who are good enough to carry out such a check are also good enough to design airplanes and would not want to work for the FAA

(Newhouse, 1983). Despite the obvious conflicts of interest, the system appears to work well.

Manufacturing standards. The FAA also sets requirements for the manufacturing processes; these standards are overseen by Designated Manufacturing Engineering Representatives (DMERs).

The manufacturing process must ensure that the finished aircraft exactly matches the designs. Each part, assembly, etc. is carefully inspected and any problems are documented and corrected. Many parts must be traceable; it must be possible to determine exactly what parts are on each airplane. Maintaining this control of the configuration requires careful and thorough record keeping.

The organization is what (Roberts, 1990) calls a high reliability organization. Quality control procedures account for about 15% of the airplane's cost (Newhouse, 1983, p. 94).

Certification. Before a new model of plane can be used, the design must be certified. Certification is a legal process that provides evidence that a certain airplane has been inspected, tested and accepted by the FAA. To certify a plane requires extensive tests. Major modifications to the design, such as adding different engines, requires recertification. For a new plane, the certification process takes about one year and may cost on the order of $100 million.

1.2.3 Market environment

The market for aircraft is highly competitive, with only a small number of buyers and sellers.

Low production rate. The total number of jet aircraft manufactured is quite small. Boeing, for example, has made more jet aircraft than everyone else combined and delivered its 5 000th jet aircraft in only 1986. Airbus manufactured 71 planes in 1988 and planned to build 101 in 1989. As a result, building airplanes is very much a batch process; component parts are manufactured in lots of ten rather thousands.

Suppliers. There are only three major manufactures still making commercial jet aircraft: Boeing and McDonnell-Douglas, two American companies, and Airbus, a consortium of European aerospace companies. In 1987, one estimate gave Boeing 51% of orders placed, Airbus 30%, McDonnell-Douglas, 17% and British Aerospace and Fokker, 2% between them ("Civil aerospace," 1988, p. 7).

Buyers. On the demand side, there are again only a small number of buyers, the major airlines. Every order is important and the manufacturers compete strenuously for each sale. The airlines have become highly skilled at playing the manufacturers against one another for the best prices on aircraft. In the past, airlines would go as far as to spread their orders among companies in order to ensure a level of competition that would hold down prices.

Changing market environment. What is important to buyers changes as the environment changes, but because of the long product development cycle there is a significant lag in the ability of the manufacturers to meet those demands.

For example, the planes now in production were mostly designed in the early 1980's when fuel was 40% of the cost of running an airplane and were therefore designed with fuel economy in mind. Today, however, fuel is only about 16% of the cost; now the cost of financing the plane is about half the total cost ("Civil aerospace," 1988, p. 8). As a result, airlines today mostly want cheaper planes and innovative financing.

Furthermore, the tax laws have changed to make owning planes less attractive. Therefore, airlines now mostly lease the planes. Many airlines have sold their planes to consortia of banks which lease them back. About 20% of the planes sold are bought directly by leasing companies ("Aerospace," 1989). In general, leasing companies are much less interested in technical details of planes and much more so in financial arrangements.

During the last several years, air travel has grown rapidly. Airlines, by and large, have done well and have been able to afford to replace aircraft. As a result, there is currently a long lead time for purchasing planes; the backlog of

orders for some models stretches out to 1994-1995. The total backlog is $54 billion or a little more than 3 years production.

The airframe companies are reluctant to increase production or invest heavily in new facilities, however, because with a downturn in the economy, the orders may disappear. (As I write this, the economy is beginning to move into a recession and several airlines are on the verge of bankruptcy, suggesting that these fears were not unfounded.) Also, hiring new employees disrupts learning curve, since resources must be spent on training instead of on building airplanes.

1.2.4 Support

Finally, support is an important part of the product. An unusable airplane earns no money for the airline and can seriously disrupt their schedules. The airlines demand assurance that their investment is protected. To meet this demand, aircraft manufacturers have facilities all over the world; if a plane has a problems, company will send parts, service teams, etc. An out-of-service airplane has the highest priority for parts or manpower.

In addition, an airplane must be maintained by the airlines to conform with the specification and with federal regulations. A large part of the profit a manufacturer can expect from a sale comes from selling spares and service over the twenty-year life span of the aircraft.

1.3 Characteristics of the organization

1.3.1 Structure

Airplanes, Inc. is organized around products; each model of airplane is developed by a new project organization. These projects use the resources of the various departments, including engineering, manufacturing, materiel, fabrication, etc. in a matrix fashion. For example, the engineering division supplies the necessary man-power to staff a new airplane project as it goes through the various stages of the development process.

1.3.2 Formal controls

At Airplanes, Inc., a heavy emphasis seems to be placed on formal control mechanisms. A large planning group develops plans for each step of the design and assembly of each aircraft; an important part of making a change is planning the change and changing the plan. One interviewee claimed that project engineers are forbidden to talk directly to workers in the plant (C, p. 95); if they need input, they can ask for a formal meeting. Manufacturing problems are handled by a formal liaison function that represents engineering to manufacturing.

In part this lack of communications is due to a lack of interest. Historically at least, engineering did not look for much input from manufacturing; manufacturing took whatever engineering designed and built it.

A major reason for this emphasis on formal procedures is the high level of turnover. People move around as the work load on projects rises and falls. For example, building the first of one plane model took about 12 million man hours of engineering effort; building the second one took only 2000 man hours more. Many of the engineers involved in the initial design were redeployed elsewhere. As a result, the process can not rely on individuals, since those individuals likely will not be there over the long term (C, p. 99). There is a belief that a large proportion of failures are due to informal procedures and dependence on individuals.

Second, for regulatory reasons, all decisions must be documented. This again stress the use of formal procedures.

2 The engineering development process

To provide a background against which to understand the change process, I will first briefly describe the process of developing a new airplane and the on-going manufacturing process.

2.1 New airplane program

Developing a new model of airplane is an enormous undertaking that is done infrequently. Boeing, for example, has developed 6 models of commercial jet aircraft, of which four are still in production, the 737, 747, 757 and 767; the 777 is reportedly in development. Since 1972, Airbus has developed four models of airplanes, the A300, A310, A320 and A330, A340; McDonnell-Douglas currently sells two, the MD-80 and MD-11.

Developing an entirely new aircraft is extremely expensive. One source estimated that to develop a new medium-sized aircraft would cost at least $2 billion ("Civil aerospace," 1988) and to design the new engines, $1.5 billion. As this figure approaches most manufacturers' entire net worth, airframe manufactures must literally "bet the company" each time they decide to develop a new plane. More often, the manufactures choose to develop a variant of an existing plane, e.g., by lengthening (stretching) an existing model to add more seats, making a freighter version, etc. The 747, for example, is offered in 11 variations (Simpson, et al., 1988).

2.1.1 Program definition

In the initial open ended stage of development, the manufacturers attempt to determine what kind of plane to build. During this phase many possible airplanes will be proposed, with various numbers of seats and range. Engineers look at what new technologies will be available and estimate the development and production costs of each design. Marketing personnel use econometric models to determine the likely market for planes and talk to the airlines to assess their future needs and get their reactions to proposed designs.

By the end of this phase, the development team creates a definition of what technologies to use, the size of plane, description of the major components, etc., and produces a document indicating the various requirements that the design must meet (e.g., statutory requirements, airworthiness requirements, the ability to operate under particular conditions, such as altitude, temperature,

length of runway) and other goals the design should attempt to achieve (e.g., maximum takeoff weight).

2.1.2 *Cost definition*

The new plane moves into the next phase when the project team gets the go ahead from the company to form a new aircraft program and begins the shift from research to the development of an actual plane. In this phase, the designers settle on one particular configuration and begin setting the schedule for designing and producing the first plane.

The engineers begin the detailed design and development of the plane. The structures group does a preliminary layout of some portion of the aircraft; then the various systems groups (such as hydraulics, engines or avionics) decide where they want to put their systems.

Several interesting techniques are used to coordinate this process. Systems groups use Coordination Sheets to request changes to the design, e.g., to request a hole in a structural element to route a pipe.

Mockups are used to support a variety of design decisions. Simple mockups are used by engineering to allocate space among design groups or to demonstrate designs ideas to management or customers. More detailed mockups are used to develop the detailed designs and to act as demonstrators for customers. Finally, full scale detailed mockups are used by manufacturing to develop manufacturing information for fabrication and installation of tubing, wires, insulation blankets, etc.

Where two parts that fit together are designed by different groups, the groups first develop an Interface Control Drawing that shows what each side of the interface should look like. These are used, for example, between different body sections, between the galleys and body, etc.

As the plane is developed the company begins looking for initial orders for the plane from what are called launch companies. These companies agree to buy the new plane; in return, they may get a price break or earlier delivery than

other airlines. At the end of this phase, the company commits to producing the plane.

2.2 Overview of production process

In order to provide a context for describing the change process, I will now briefly describe the production process and indicate where changes can occur.

2.2.1 *Customer contracts for a plane*

Aircraft are highly customized for each customer; each plane has a unique configuration. Development of a customer's configuration starts with the standard aircraft. Many options need to be specified to define a usable aircraft which meets the customer's requirements.

To accommodate the level of customization needed, the design of the airplane is conceptually divided into two parts: basic and variable. All customers get all parts released for the basic airplane, for example, the wings and most of the body of the plane. The basic parts are capable of accepting a variety of variable parts, for example, the arrangement of galleys or lavatories, or the avionics or engines used, which can differ depending on the customer's preference.

In parallel with the negotiation on the aircraft configuration, the customer and Airplanes, Inc. negotiate and price and delivery date. The price charged may have little connection with how much it actually costs to build the airplane. Non-standard changes require additional engineering work and so might cost more to implement or delay the delivery of the plane. Airplanes, Inc. also guarantees the maximum takeoff weight of the plane and various aspects of its performance.

2.2.2 *Plan is developed*

The customer specification is used to develop a basic statement of work, laying out everything that needs to be done to deliver the plane. Based on this statement of work, the company develops a plan indicating the steps necessary to manufacture and deliver the plane.

After the plan is developed, changes may still be made to the configuration, except they now require changes both to the design and the plan. Changes are made both on customer request and for on-going product improvements.

2.2.3 *Plan is executed*

To actually build the aircraft, each group carries out their parts of the plan.

Engineering release. Engineers work according to the engineering schedule which indicates what engineering data is needed and the date by which it must be released.

Engineering data includes the drawings of each part and assembly in the aircraft, material lists indicating what raw materials and standard parts will be needed so they can be ordered in advance, parts lists, tooling information, installation drawings and test data. The engineers must also consider changes necessary to the mockups as well as production planes. Engineering may also release Production Memos telling manufacturing to stop making parts that will be cancelled by a new release.

If the customer has taken all standard options, then the engineer process is straightforward, since the engineer can simply indicate that the existing engineering data should be used for the customer's plane.

For a new design, the engineer first prepares layouts showing general arrangement of parts and has the layout approved. A drafter then prepares the detailed drawings of the parts and the engineer signs off that the designs are correct. New drawings may be reviewed to ensure they meet interfaces.

When a piece of work is complete, the project engineer sends the data and a drawing data sheet to Engineering Data Control. The data sheet lists all engineering releases necessary to accomplish each piece of work. This list is used by the downstream groups as a check-list to ensure they take all changes in to account. For a particular change, the engineer may request that advanced copies of a new drawing be sent to particular organizations.

Engineering Data Control checks the data to ensure that it meets the appropriate drafting standards and then marks the items in the schedule as done. Drawings are copied and distributed to the downstream groups. Parts list information is added to a computer system and used to create a Bill of Materials for each plane.

Configuration review. If an aircraft is the first of a model to be delivered to a customer, then Customer Engineering and Project Engineering review the drawings to ensure that the plane being designed meets the customer requirements (C22, p. 3-4).

Manufacturing. The final assembly of the plane is also done according to plan. The plan starts with the manufacture or procurement of unit parts. Parts can take as long as several years from the time they are ordered to when they are available. The individual parts are then assembled into subassemblies and subassemblies into assemblies. As the plane moves down the assembly line in final assembly, it passes through a number of stations, staying at each station for about a day (depending on the model). At each station, a certain set of assemblies are installed and tested.

In the plant, each worker has a band, which lists a particular sequence of tasks they need to do. When the worker finishes a task, they cross off what they've done. A new set of instructions are issued for each plane that passes through the station. For each task listed there's a sheet called an O&IR (Operation and Inspection Record) describing which parts are necessary and then all the steps that have to be done. It used to be that the factory worked off of the drawings for each plane. Now they work off O&IRs. After doing the same job for a while, they might not have to read the O&IR each time.

The parts listed are either standard small parts, like bolts, etc., which are pulled from bins scattered around the plant floor or parts which are picked by stores and put in a special box for that task. When a worker starts a job, he gets the bin of necessary parts. When he finishes the task, he stamps it done and puts a sheet out for the inspector. The inspector checks that it was done correctly and then stamps it complete as well.

If the inspector finds a problem, then he or she puts a rejection tag on the work describing the problem. A special group of engineers, called liaison engineers, work in the plant to resolve these problems. Most often the problem is due to a manufacturing error. In these case, the part can be reworked or, if the problem is inconsequential, accepted as is. Some of the time, the problem is due to an engineering error, in which case the liaison engineer can change the drawing or refer it back to the project engineer. These design changes are made without being committed by the change board, essentially because plan is not modified.

There are also a number of "greenlines," which are descriptions of work needed to resolve problems noted on rejection tags. These are not part of the standard O&IR since they are not part of the released plane, but some do appear on the employees' bands. (Some rejection tags remain for hundreds of airplanes; one had been in the factory for 2 years. That was a case where a subdivision had been building the parts incorrectly and the assembly plant was fixing them, and the subdivision had not been able to correct the process.)

2.2.4 *Plane is delivered*

Eventually the plane is finished. After testing, it is delivered to the customer. Even after plane is completed, changes can be made. Changes may be made to already delivered plane to incorporate changes which the customer requests or important changes which to fix problems that limit the capabilities of the aircraft or to meet new regulatory requirement.

3 The change process

Now that we have seen the initial design process, we can discuss how changes fit into it. More so than for other products, it is difficult to separate the change process from the production processes. In some sense, Airplanes, Inc. sells change management.

3.1 Dimensions of changes

Changes differ along (at least) two dimensions: when the change is made and who initiates it.

When change is made. Some change are made as part of the design for a particular customer's plane while others are made during production or even after the plane has been delivered. In this chapter, I will not discuss changes made after delivery, although many of the processes discussed are also applied to these changes.

Originator of change. Changes may be initiated by both the customer and Airplanes, Inc.. Customer requested changes are typically paid for by the customer; Airplanes, Inc. initiated changes are paid for by the company and are typically made to improve the design or make the plane cheaper or easier to build.

3.2 Goals of the change process

Design changes are necessary to meet customer demands for changes and to improve the design or manufacturability of the plane. However, the change control process must ensure that the company retains control of the configuration (i.e., knows exactly what design and components were used for every part on every plane built) and to control costs, by balancing the cost of the changes against their benefits.

3.3 An informal description of the change process

There are many similarities in change processes for different kinds of changes. A primary difference is when the change is made.

3.3.1 Contract negotiation

Many changes are made as a part of the initial contracting process. A potential customer negotiates with Airplanes, Inc. to purchase planes. These

negotiations include the cost of the plane, delivery date, performance guarantees and the details of the airplanes' configuration.

Negotiations about the configuration start with the basic specification for the plane. The customer submits change requests for everything beyond the standard aircraft, e.g., to choose various options or to request particular changes.

Standard options are changes that already engineered and priced and the customer simply decides whether or not to take them, e.g., a redundant warning system or additional piece of avionics (Newhouse, 1983). In some cases, there may be no default option and the customer must make a choice, e.g., which manufacturer's engines to use or the exact arrangement of the passenger compartment.

Novel change requests must be evaluated before company agrees to offer them and the price, effect on performance, etc. negotiated with the customer. Often these are unique options developed for a particular customer. For example, one airline has its own standard toilet drain connection and requires it on planes it buys. Sometimes a customer-initiated change will be adopted as a standard option and offered to all customers.

In addition, the airlines can choose to not have certain Airplanes, Inc. initiated changes made on their planes. (One reason for not taking a change is to reduce the amount of variation between airplanes in a customer's fleet.) Changes the customer decides not to take must be backed out using a Change Request, since the changes have become part of the basic release.

Customer change requests are handled in a two phase process:
(1) determine if the change can be offered and negotiate the effect on schedule, price, performance, etc. with the customer; (2) if the customer accepts the proposal, implement the change.

The customer submits a change request to customer engineering indicating what change they want. Customer Engineering writes a narrative Change Request Work Statement (CRWS) describing generally what work is required. This work statement may provide only enough detail to allow

estimates of the cost and work necessary, such as the major parts to be replaced or revised, new items to be added or reference to similar changes already made. In some cases a fully detailed work statement may be generated initially, indicating exactly what work will be done at a detailed level (C42, p. 3-12). The customer engineer may check the work statement with the responsible project group.

The customer engineer then sends an alert message to the affected design groups asking them to classify the change. Changes are classified into one of four categories: (1) a specification change, one that changes the language of the specification but not the plane; (2) a negotiated change, one that changes something about plane that needs to be negotiated with customer, such as cost, performance, weight, etc.; (3) a study item, a change that requires more investigation; or (4) rejected. The project engineers inform the Customer Engineer by phone of the classification.

The Customer Engineer then distributes copies of the work statements to other groups for their evaluations. One copy goes to the Program Management Office (PMO). The PMO chairs a Change Request Review Board, which includes representatives from planning, materiel, etc. This groups decides whether or not the company can offer the change in the time available.

Other groups determine what they will have to do to implement the change. For example, the testing group determines what kind of testing will be required for the change in consultation with the Designated Engineering Representatives.

The Engineering Cost and Schedules group estimates the engineering hours needed, which is used by the Finance Group to estimates the total cost of making the change. The cost is used by the Pricing group to set the price for the change.

The Weights group estimates the effect of the change on the weight of the plane. The entire change package is approved by the Customer Engineering Management.

The change package is sent to Contracts, who handle all negotiations with the customer. Contracts presents the change proposal to the customer, who then decides whether to take the change or not. Accepted changes are incorporated in the contract and Customer Engineering records the necessary changes in the Customer Detail Specification.

3.3.2 *Initial planning*

At some point, the customer signs the contract to purchase the planes and is given a particular line position. The plane in that position will be built to meet the customer's configuration.

When the contract is signed, the Contracts group tells the Program Management Office, which issues a Program Directive telling everyone to do what is necessary to deliver the planes. Engineering Business Management then issues a Program Implementation Memo (PIM) to authorize the various engineering groups to release the necessary drawings for the plane. Customer Engineer distributes the Configuration Project Memo to all engineering groups indicating the exact configuration of the plane.

Engineering Business Management sets a preliminary work statement based on the past plans and the changes requested and sends them to the project engineers along with a deadline for the Change Request Work Statements (CRWS). The project engineers then reviews the configuration memo and develop a detailed work statement indicating what they need to do.

Engineering Work Statement

Given the customer configuration, the project engineers develop an Engineering Work Statement showing in detail what engineering needs to be done for the new customer and the dates the various pieces of engineering data will be available.

Based on the preliminary work statement, manufacturing engineering develops a Commitment Development Schedule (CDS) and a Manufacturing

Work Statement (MWS). The CDS is a plan showing the steps necessary to build the plane and the order of the steps.

Tooling develops the tooling design and fabrication flow times and other groups determine the necessary times for making and buying parts. Industrial Engineering takes flow times for various stages and develops demand dates, that is, the dates by which they need to receive the engineering drawings. The various groups then negotiate any discrepancies between the dates to arrive at a final schedule.

Engineering Administration publishes the schedule and enters the engineering release dates in the Engineering Scheduled Work Report System (ESWR).

3.3.3 *Changes after the plan is complete*

Changes are also made after the contract has been signed but before the plans are built and delivered. If the planning has not be completed (e.g., for an aircraft that has been ordered but for which the delivery date is well in the future) these changes can be handled as above. If the plan has already been made, then the change requires replanning.

Customer requested changes

After contract is signed, the customer can still request changes to the configuration. For example, some of the requested changes may not have been resolved by the time the contract was signed. However, any additional cost of the change must be negotiated. These change requests are also handled in two stages.

As before, the customer submits a change request to the Customer Engineer assigned to that Customer. The Customer Engineer prepares a narrative work statement indicating what changes are necessary in consultation with the Project Engineers as necessary.

The Engineering Change Control group issues an action memo describing the change that needs to be made which is sent to the Test Integration and

Weight groups, to staff engineers, etc. for their inputs. Based on the estimated work, Finance and Pricing develop a price for the change and the change document is approved by Customer Engineering Management.

Unlike earlier changes, the change is reviewed for offerability by the Manufacturing Change Board. The Change Board is chaired by Manufacturing and includes representatives of all affected groups. There are different boards for different airplane models. The meetings I attended were attended by representatives from the factory, the purchasing department, Manufacturing Engineering, Industrial Engineering, Quality Control, a subassembly plant, Engineering Change Control and the Change Board Chairman (a full time job).

The completed change package is then given to Contracts for negotiation with Customer. If the customer accepts the change, Contracts amends the contract and Customer Engineering amends the configuration memo.

Engineering Change Control then requests the engineering release dates from the engineers to complete the Engineering Change Memo and the ECM is submitted to the change board for commitment to a particular airplane. (The commitment process is described below.)

On-going design improvements

Many changes are initiated by the Airplanes, Inc. company. Such changes start with a requirement for a design change going to the prime design group. For example, whenever an airline has a problem with an plane in service, a report from the airline is sent to Customer Service Engineering group and eventually makes its way to the design engineer responsible for the affected part.

The project engineer develops a solution to the problem and prepares an engineering change memo indicating the reasons for change, change priority, effectivity and information necessary for commitment by change board. The engineer also notes which other engineers and downstream groups are affected by the proposed change. The form is then sent to Engineering Change Control for processing.

Changes are assigned one of three priorities (C10, pp. 4.20–21). Compulsory means the change must be made without regard to program impact. Such changes include fixes to safety or reliability problems that limit the operation of aircraft or to meet government regulations or contractual requirements. Urgent changes are technically desirable and should be implemented as soon as possible, even if this requires out of sequence work. Routine changes are desirable but do not justify out-of-sequence work (see below).

ECC sends the handwrite to the affected groups, possibly adding other groups who they believe may be affected. These groups prepare work statements indicating what they need to do to implement the change and return them to ECC. ECC then issues a tip sheet and sends it to various staff groups, such as finance, weights, and staff engineers as above. These groups also prepare their work sheets and return them to ECC.

The change is then presented at a weekly director's meeting. The responsible project engineer presents the details of the change to the director of engineer for the plane. Based on the presentation and the additional data collected, the engineering manager approves or disapproves the change.

If the change is approved, ECC issues a preliminary change memo and distributes it to the other groups. The change is then taken up at the Superboard meeting, where it is discussed. The Superboard is a weekly meeting of directors, e.g., director of engineering, of manufacturing, etc. and is attended by about 30 or 40 people. This groups hears presentations of all new changes that week along with a financial analysis saying how much the change costs or saves. I attended two meetings of the Superboard. One week there were 4 and the other 5 changes total for two models of aircraft.

According to some sources, the Superboard considers both Airplanes, Inc. and customer initiated changes, but no customer changes were discussed at the meetings I attended and it is unclear where in the customer change process this is supposed to happen.

The Superboard tells the Change Board how important the change is to implement. They may do nothing; they may decide to hold the change for an additional piece of information; or they may decide to kill the change completely.

Implementing the changes

Customer requested and internally initiated changes are implemented in essentially the same way.

Once the change has been approved, Engineering Change Control ask the project engineers for an Engineering Release Schedules (ERS) indicating when the engineering can be done. These and the the detailed Work Statement are sent to the Manufacturing Change Board for commitment to a particular aircraft.

The change boards meet daily to discuss all open changes. On the two days I attended, one board had 23 and 24 changes for which commitment was being developed and 23 and 12 for which the commitment was being reconsidered. No new changes were presented. Of this number, only a few are discussed (7 or 8) and fewer still are committed.

Committing a change means identifying a particular aircraft on which to implement a change and getting each group to commit to the necessary delivery dates for their pieces of a change. The board develops a modified plan indicating when all the pieces of the change will be ready and which planes will be affected. As the plane gets closer to completion, the scale of changes that can be made is reduced.

Scheduling a change. Generally, Airplanes, Inc. does not wait to make a change, since there are no model years or other logical break points in the assembly process. If the change makes sense, then it is implemented as soon as possible.

However, some changes may be implemented either for all or none of a particular customer's fleet, to avoid producing incompatible planes. For example, if an engineer decides to change a fitting on a pipe, they may not

implement the change on specific airplanes to avoid selling an airline two planes with different fittings.

A particular change has an effect at a given station on the assembly line. Ordinarily, changes are committed to a particular plane based on which plane will be at that station on the day the new parts are ready. These kinds of changes are called in-sequence, because the work is done in the planned sequence.

Some changes (in particular, customer requested changes) are required for a particular customer's plane, in which case the change has to be made to that plane. In this case, the change may be done out-of-sequence, meaning that parts that are supposed to be installed in one station may instead be installed further down the line. As an extreme case, important changes may have to be made to all planes, including planes that were already sold and delivered.

Making out-of-sequence changes is very expensive and is generally avoided. Doing an installation out-of-sequence might make it harder to install the parts, since parts installed later might obstruct access. Alternately, it may be impossible to install some of the later parts until the earlier parts are installed. In any case, out-of-sequence work makes testing the work difficult or impossible, disrupts normal work in the later station and requires extra workers.

To schedule a change, the Change Board sends the work statement to the planners. The planners create a Commitment Development Schedule (CDS) (C10, p. 4-29) that indicates each event and the order of events that must take place to accomplish the action. This plan is used as a basis for negotiating when each group must complete its work.

At any point, a change is held waiting for a particular group to obtain some information. For example, all changes are initially held for Planning to develop the CDS. A change may be held while Materiel gets a quote or a lead time from a supplier, for engineering review, etc. On days I visited, most of the changes were waiting for additional information; many were in fact past due.

Once the availability dates for each part of the plan are determined, the change board issues the Change Commitment Record (CCR) (C10, p. 4-30). This

is a form signed by the representatives of each group saying that they will do the necessary work. The form also records to which airplanes each part of Engineering Change Memo is committed.

Engineering Change Control then issues the committed ECM and the engineering release dates are entered in to ESWR as above. The downstream groups also implement their parts of the changes. The entire plan is monitored by a change compliance group that checks that committed events are completed on or before due dates.

Recommitment. The plan is made based on the preliminary engineering change memo which is issued before the drawings are available. Sometimes when the change is finished, it turns out that the changes are different from expected. Since the change is scheduled as tightly as possible, any delay in a single step in the plan may affect the outcome. In this case, the change must be recommitted. The affected group can submit a recommitment evaluation request (RER) (C10, p. 4-32, 4-33) which is considered by change board. The board can change the plan for the new circumstances and recommit the change.

3.3.4 Changes during production

Changes can be made to the airplane even during the production process. Some of these changes are in response to customer requests and are processed in more or less the same way as above, but on a smaller scale, since the time available to make the change is so short.

Others are minor changes made in the plant. These changes must still be carefully documented. A major source of these small changes are problems noticed by personnel in the plant. These problems are usually handled by a group of Liaison engineers who work in the plant.

Rejection tags

Quality control inspects each part, assembly, installation to look for disagreements between the drawings and plane or other problems (e.g., two parts rubbing together, a slight dent in a piece of sheet metal, etc.). These

problems may also be reported to Quality Control by the customer's own inspectors who work in the plant or by the FAA inspectors.

Quality Control writes a Rejection Tag for the part stating what is wrong. An interviewee who was responsible for Liaison engineering for two plane models said that there were about 7000 rejection tags in a typical week, with some weeks having as many as 11000.

These figures were for planes that had been in production for at least several years. Rejection tags are used as one measure of how far along the learning curve a plane has progressed. On the first plane of a model has a large number of rejection tags as the final problems with the design are worked out. For each customer introduction, the number of rejection tags again goes up slightly.

The liaison engineer reads the rejection tag and inspects the parts to determine what fix is needed, describes the required change on the rejection form and signs the form. The rejection tag then goes to Manufacturing to be implemented.

According to one interviewee, 97 to 98% of all rejection tags are due to manufacturing errors, most commonly misdrilled holes. The Liaison Engineer may decide to scrap the part and start over, to accept the part as is or rework the part in some other way. Finding an appropriate disposition for the parts is especially important if the parts involved are particularly complex or expensive. If the parts must be reworked, then a rework order is prepared, which eventually results in greenlines for particular stations. In any event, records are kept to indicate what was done for each problem; the customer eventually receives copies of all rejection tags.

Engineering problems

About 2 to 3% of the problems are due to engineering errors. In this case, the drawings must be changed in order to eliminate the problem on future airplanes.

In addition to errors noted on the rejection tags, Liaison engineers handle requests for changes from several other groups. For example, employee suggestions are sent to the Liaison engineers for evaluation and possible implementation.

Manufacturing can request changes to the design using a form called an Engineering Liaison Request. Such a request may be used for several reasons, such as to point out an engineering error, for example, a reference to an incorrect part number or to request permission to substitute materials or make minor changes to facilitate tooling or the production process.

Subcontractors may similarly request changes, but all communications from a subcontractor must be sent by way of the Materiel department. The Materiel department then forwards the requests to the Liaison engineer for evaluation.

In all these cases, the liaison engineer assess the requested fix on behalf of the engineering department. If the change seems inappropriate, the liaison engineer may reject it. For small changes, liaison engineers can change the drawings themselves. Liaisons can make changes when purchased or buyer furnished equipment is not affected, the new parts, materials or processes are readily available, where airworthiness or FAA is not involved, where purchase agreements do not require customer approval and when engineering and manufacturing agree on the change. However, a change must be approved using the full change process when the fit or function of purchased equipment or manufactured parts is affected, buyer furnished equipment is affected or new material or processes are required, or when airworthiness is affected.

The Liaison Engineer may consult with the project engineers if there are any questions about what to do.

Liaison makes change

To issue a change, the Liaison Engineer has a Drafter prepare an Advanced Drawing Change Notice (ADCN) that changes the drawing. An ADCN is a single sheet of paper that makes some small change. It is added to

the front of the drawing instead of changing the drawing itself. ADCNs are incorporated when the drawing needs to be changed, i.e., when a bigger change is made. The ADCN is then sent to Engineering Data Control which releases it.

Liaison engineering can do new drawings for all of the detailed parts, about 50% of the assembly problems and only about 33% of the problems in final assembly because of increased complexity of problems.

The change is reviewed by Manufacturing Engineering Representative prior to release. The Manufacturing Engineer Representative considers the schedule and availability of resources and determines the change incorporation point. The representative prepares the Liaison Change Commitment Record that indicates when the engineering, replanning, etc. will be ready and which airplane the change will affect.

Some changes do not have to committed to a particular plane; these are changes do not change the parts but perhaps fix errors in release, such as referring to an incorrect part number.

Some do not require commitment on the drawings but are committed to a particular plane by manufacturing, using a Liaison Change Commitment Record. This is the case when the parts made according to the rejection tag are satisfactory but the parts that have not yet been installed or assembled need to be reworked.

Other changes require commitment to a particular airplane on the drawings. This is the case when the parts are changed in a way that must be recorded and the airplane and parts must comply with the change.

Miscellaneous changes group

If the change is not one that the liaison can make, then the liaison forwards the problem to the project using a Liaison Design Action Request form (LDAR). These change are forwarded to a Miscellaneous Changes Group which schedule them in the Engineering work schedule.

The miscellaneous problems group also get requests for change from several other groups. For example, other drawing processing or material ordering groups may send an Engineering Problem Items form to notify the engineers of parts listing errors or problems, e.g., failure to list a part on the drawings in the parts lists.

Project engineer

The engineer works on the change as required by the engineering work schedule. The project engineer may also decide that the change is significant enough that it should be made as a controlled change, as described above.

3.4 Perceived problems with the change process

As usual, one perception is that there are too many changes being made. A second concern is that various groups, in an effort to meet delivery deadlines, agree to schedules for changes that they can not meet, thus further disrupting the process.

The high volume of work is a problem throughout the aerospace industry and may be leading to quality control problems, such as those reported by (Fitzgerald, 1989).

4 Models of the change process

The information-flow and intentional models are included as appendices to this chapter. The full model has 25 types of actors, described in Table 6.1.

Table 6.1. *Actors types for the Airplanes, Inc. model.*

Type of actor	*Function in change management process*
Change review board	determine if proposed change can be offered in time allotted
Contracts	official channel for communication with customers
Customer	decide features are necessary evaluate and accept or reject change proposals
Customer Engineer	determine work necessary to implement requested change
Customer Engineering Manager	approve proposed changes
Designated Engineering Representatives	check that proposed designs are technically acceptable approve test plans
Division Finance	given statement of work and engineering hours, calculate cost to make a change
Drafter	given designs, prepare detailed drawings
Employee Suggestion Unit	forward suggestions from employees to liaison engineer for evaluation
Engineering Business Management	add events to Engineering Scheduled Work Report negotiate engineering release dates with Manufacturing Engineering
Engineering Change Control	given statement of work, cost of change and effect on weight, prepare change memo represent engineering for change control

Engineering cost and schedules	given engineering statement of work and test plan, determine cost of engineering hours
Engineering Data Control	check that drawings meet standards and forward to downstream groups
Engineering Manager	approve layouts and proposed changes
Liaison Engineer	given a problem, determine scope of change necessary make minor changes to drawings
Manufacturing Change Board	decide if a change proposed after the contract has been signed can be offered in the time allotted given dates, commit proposed change to particular airplane
Manufacturing Engineering	identify problems with drawings given workstatement, determine dates when engineering drawings are needed schedule implementation of small changes
Miscellaneous Changes Group	add work to Engineering Scheduled Work Report
Planning	given a workstatement, prepare a plan, showing necessary steps and precedence relationships
Pricing	given the cost of a change, determine what price to charge
Project Engineer	determine that a change is necessary given change, prepare detailed description of work necessary given description of work, prepare release schedules given configuration and schedule for work, prepare designs and other engineering data
Quality Control	check that drawings and parts, assemblies or installations match
Subcontractor	identify problems with drawings

Super Board	approve proposd changes
Weights	determine effect of a change to weight of airplane

6-1

INFORMATION-FLOW MODEL FOR AIRPLANES, INC.

#	Sender	Message	Recipient	Actions taken
3	Customer Engineer	*Change request work statement*	Change review board	decide if change can be offered in time available send *Offerability* message to Contracts [8]
8	Change review board or Manufacturing change board	*Offerability*	Contracts	wait for *Offerability, Change Request Document* and *Price* messages when all have been received, if change can not be offered then send *Change request rejected* message to Customer [30] otherwise, send *Proposal* to Customer [15]
17	Customer	*Change acceptance*	Contracts	add Change Request to contract send *Change acceptance* message to Customer Engineer [19]
18	Customer	*Change rejection*	Contracts	file change for future reference
21	Customer	*Contract acceptance*	Contracts	send *Contract acceptance* message to Program Management Office [22]
10	Customer engineer	*Change Request Document*	Contracts	see [8]
16	Pricing	*Price*	Contracts	see [8]
1			Customer	determine requirements send *Change request* message to Customer Engineer [2]
20			Customer	decide to purchase airplane send *Contract acceptance* message to Contracts [21]

15	Contracts	*Proposal*	Customer	decide whether or not to accept the change send *Change acceptance* [17] or *Change rejection* [18] message to Contracts
30	Contracts	*Change request rejected*	Customer	possibly resend *Change request* message [2] possibly drop change request
19	Contracts	*Change acceptance*	Customer Engineer	add change to Customer Configuration document if contract has been signed then send *Change acceptance* message to Engineering Change Control [44] send revised *Customer Configuration* message to Project Engineer [48]
2	Customer	*Change request*	Customer Engineer	prepare *Change request work statement* if necessary consult with Project send *Request for classification* message to Project Engineer [26]
47	Customer Engineering Manager	*Change Request Document*	Customer Engineer	if approved, then send *Change Request Document* message to Contracts [10] otherwise, revise the *Change Request Document* and resend it to the Customer Engineering Manager [29] or possibly drop it
23	Program Management Office	*Program Directive*	Customer Engineer	send *Customer Configuration* message to Project Engineer [48]

27	Project Engineer	*Change classification*	Customer Engineer	send a *Change request work statement* message to Test Integration [4], Engineering Cost and Schedules [6] and Weights [7] if the change was requested before the contract was signed, send a *Change request work statement* message to Change Review Board [3] otherwise, send a *Change request work statement* message to the Manufacturing Change Board [43] and to Engineering Change Control [45]
62	Test Integration	*Test plan*	Customer Engineer	see [12]
12	Weights	*Weight*	Customer Engineer	wait for the *Test Plan* and *Weight* messages add weight information and test plant to *Change Request Document* message send *Change Request Document* message to Customer Engineering Manager for approval [29]
29	Customer Engineer	*Change Request Document*	Customer Engineering Manager	approve or reject entire change package send *Change Request Document* message to Customer Engineer [47]
9	Test integration	*Proposed test*	Designated Engineering Representatives	return *Comment* message approving or disapproving proposed test plan [28]
11	Engineering cost and schedules	*Cost*	Division Finance	determine total cost of offering change if the change was requested by the Customer, send *Cost* message to Pricing [14] otherwise, send *Cost* message to Engineering Change Control [5]

40	Project Engineer or Liaison Engineer	*Design*	Drafter	prepare final drawings return *Drawings* message [41] or [73]
84			Employee Suggestion Unit	send a *Suggestion* message to the Liaison Engineer [85]
54	Engineering Change Control	*Committed Engineering Change Memo*	Engineering Business Management	add new dates to the Engineering Scheduled Work system send *Engineering Scheduled Work* message to Project Engineer [36]
35	Manufacturing Engineering	*Proposed schedule*	Engineering Business Management	negotiate discrepencies between original engineering release dates and engineering demand dates send *Engineering Scheduled Work* message to Project Engineer [36]
24	Program Management Office	*Program Directive*	Engineering Business Management	send *Implementation Memo* message to Project Engineer [25]
31	Project Engineer	*Engineering workstatement*	Engineering Business Management	compile workstatements send *Engineering workstatement* message to all Project Engineer [32]
33	Project Engineer	*Engineering Release Schedule*	Engineering Business Management	compile the Engineering Release Schedules send the *Engineering Release Schedule* message to Manufacturing Engineering [34]
44	Customer Engineer	*Change acceptance*	Engineering Change Control	send *Change request work statement* message to Project Engineer [49]

45	Customer Engineer	*Change request work statement*	Engineering Change Control	not sure generate *Change Action Memo* message and send to Manufacturing change board [46]
5	Division Finance	*Cost*	Engineering Change Control	see [58]
65	Engineering Manager	*Preliminary PRR*	Engineering Change Control	if approved, then send *Preliminary PRR* message to all affected groups, including the Super Board [66] otherwise, revise the *Preliminary PRR* and resend it to the Engineering Manager [64] or possibly drop it
67	Engineering Manager	*Approved PRR*	Engineering Change Control	if approved, then send *Approved PRR* message to the Manufacturing Change Board [52] otherwise, revise the *Preliminary PRR* and resend it to the Super Board [66] or possibly drop it
53	Manufacturing Change Board	*Committment Development Schedule*	Engineering Change Control	issue the committed engineering change send a *Committed Engineering Change Memo* message to Engineering Business Management [54] and all other groups
50	Project Engineer	*Engineering workstatement*	Engineering Change Control	compile workstatements send *Engineering workstatement* messages to Project Engineer to request release schedules [32]
51	Project Engineer	*Engineering release schedule*	Engineering Change Control	send *Engineering Change Memo* message to Manufacturing Change Board [52]

56	Project Engineer	*PRR Handwrite*	Engineering Change Control	prepare PRR form send *Preliminary PRR* message to affected Project Engineer [57]
61	Project Engineer	*Engineering workstatement* and *Engineering Release Schedule*	Engineering Change Control	send *PRR Tipsheet* to Test Integration [4], Engineering Cost and Schedules [6] and Weights [7]
63	Test Integration	*Test plan*	Engineering Change Control	see [58]
58	Weights	*Weight*	Engineering Change Control	wait for *Weight*, *Test plan* and *Cost* messages add weight, test plan and cost information to PRR send *Preliminary PRR* message to Engineering Manager for approval [64]
6	Customer Engineer or Engineering Change Control	*Change request work statement* or *PRR Tipsheet*	Engineering cost and schedules	wait for *Change request work statement* and *Test plan* messages determine cost of engineering and testing necessary send *Cost* message to Division Finance [11]
13	Test integration	*Test plan*	Engineering cost and schedules	see [6]
42	Project Engineer or Miscellaneous Changes Group	*Release*	Engineering Data Control	check that drawing meets standards; if there are problems, send an *Engineering Problem Item* message to the Miscellaneous Changes Group [88] make copies and store originals send drawings and other data to downstream users (not modelled so far)

64	Engineering Change Control	*Preliminary PRR*	Engineering Manager	approve or reject entire change package send *Preliminary PRR* message to Engineering Change Control[65]
37	Project Engineer	*Parts layout*	Engineering manager	return *Layout approval* [38] or *Layout disapproval* message [39]
73	Drafter	*Drawings*	Liaison Engineer	send a copy of the *Drawings* message to the Project Engineer [78]
85	Employee Suggestion Unit	*Suggestion*	Liaison Engineer	if the design should be changed and the change is minor, then send a *Design* message to the Drafter [40] if the change is too big for the Liaison to work, send a *Liaison Design Action Request* message to the Project Engineer [79]
72	Manufacturing Engineering	*Engineering Liaison Request*	Liaison Engineer	if the change is not needed, return a *Change request rejected* message [83] if the design must be changed and the change is minor, then send a *Design* message to the Drafter [40] if the change is too big for the Liaison to work, send a *Liaison Design Action Request* message to the Project Engineer [79]
75	Project Engineer	*Advice*	Liaison Engineer	continue process at [70]

70	Quality Control	*Rejection Tag*	Liaison Engineer	determine what should be done to fix problem if in doubt, possibly send *Request for advice* message to Project Engineer [74] and wait for *Advice* message if the parts are acceptable, send a *Accept parts* message to Manufacturing [77] if the parts need to be reworked, send a *Rework* message to Manufacturing [76] if the parts are acceptable, but the design must be changed and the change is minor, then send a *Design* message to the Drafter [40] if the change is too big for the Liaison to work, send a *Liaison Design Action Request* message to the Project Engineer [79]
76	Liaison Engineer	*Rework*	Manufacturing	rework parts to design and continue manufacturing process
77	Liaison Engineer	*Accept parts*	Manufacturing	continue manufacturing process
43	Customer Engineer	*Change request work statement*	Manufacturing Change Board	decide if change can be offered in time available send *Offerability* message to Contracts [8]
46	Engineering Change Control	*Change Action Memo*	Manufacturing Change Board	see [43]
52	Engineering Change Control	*Engineering Change Memo* or *Approved PRR*	Manufacturing Change Board	send Engineering Change Memo message to Planning [59]

60	Planning	*Committment Development Schedule*	Manufacturing Change Board	negotiate milestone dates commit the change to a particular airplane send a *Committment Development Schedule* message to Engineering Change Control [53] and all other groups
71			Manufacturing Engineering	notice some problem with the engineering data or suggest way to facilitate production send *Engineering Liaison Request* message to Liaison Engineer [72]
34	Engineering Business Management	*Engineering Release Schedule*	Manufacturing Engineering	based on the workstatement and the customer configuration, develop the master schedule, adding times for tooling, making and procuring parts determine when engineering data is needed send *Proposed schedule* message to Engineering Business Management [35] publish plan for manufacturing parts, assemblies and final plane
83	Liaison Engineer	*Change request rejected*	Manufacturing Engineering	possibly revise and resend the *Engineering Liaison Request* message [72]; possibly drop the request
89	Miscellaneous Changes Group	*Release*	Manufacturing Engineering	review release develop Committment record return *Liaison Change Committment Record* message [90]
82	Project Engineer	*Release*	Miscellaneous Change Group	send *Release* message to Manufacturing Engineering for review [89] (I'm actually guess that this is where this happens)

88	Engineering Data Control	*Engineering Problem Item*	Miscellaneous Changes Group	if the change makes sense, add it to Engineering Scheduled Work send *Engineering Scheduled Work* message to Project Engineer [36]
79	Liaison Engineer	*Liaison Design Action Request*	Miscellaneous Changes Group	add change to Engineering Scheduled Work send *Engineering Scheduled Work* message to Project Engineer [36]
90	Manufacturing Engineering	*Liaison Change Committment Record*	Miscellaneous Changes Group	add Liaison Change Committement Record to the release data send *Release*to Engineering Data Control [42]
87	Subcontractor	*Engineering Change Request*	Miscellaneous Changes Group	if the change makes sense, add it to Engineering Scheduled Work send *Engineering Scheduled Work* message to Project Engineer [36]
59	Manufacturing Change Board	*Engineering Change Memo*	Planning	develop a Committment Development Schedule, indicating the necessary steps to implementing the change return a *Committment Development Schedule* message [60]
14	Division finance	*Cost*	Pricing	determine price of change send *Price* message to Contracts [16]
22	Contracts	*Contract acceptance*	Program Management Office	send *Program Directive* message to Customer Engineer [23] and Engineering Business Management [24]
55	Customer	*Problem report*	Project Engineer	determine that a change is necessary develop a preliminary change send *PRR Handwrite* message to Engineering Change Control [56]

26	Customer Engineer	*Request for classification*	Project Engineer	return *Change classification* message, one of Specification change, Design Change, Standard Option [27]
48	Customer Engineer	*Program Directive*	Project Engineer	use Configuration to guide design work
41	Drafter	*Drawings*	Project Engineer	check content of drawings if there are problems, return *Design* message to Drafter for more work [40] otherwise send *Release* message to Engineering Data Control [42]
25	Engineering Business Management	*Implementation Memo*	Project Engineer	review narrative work statements and prepare more detailed work statement send *Engineering workstatement* message to Engineering Business Management [31]
36	Engineering Business Management	*Engineering Scheduled Work*	Project Engineer	do work required by schedule for parts, prepare layouts of parts and send *Parts layout* message to approver (presumably the engineering manager) [37] if the work is a miscellaneous change and can be released as such, send a *Release* message to the Miscellaneous Change Group [82] if the change can not be released as a Miscellanous Change, then develop a preliminary change and send *PRR Handwrite* message to Engineering Change Control [56] for other kinds of Engineering Data, such as Engineering Advanced Material Requirements (EAMRs), parts lists, etc., send *Release* message to Engineering Data Control [42]

32	Engineering Business Management or Engineering Change Control	*Engineering workstatement*	Project Engineer	return *Engineering Release Schedule* message [33] or [51]
49	Engineering Change Control	*Change request work statement*	Project Engineer	review narrative work statements and prepare more detailed work statement send *Engineering workstatement* message to Engineering Change Control [50]
57	Engineering Change Control	*Preliminary PRR*	Project Engineer	review narrative work statements and prepare more detailed work statement send *Engineering workstatement* and *Engineering Release Schedule* messages to Engineering Change Control [61]
38	Engineering manager	*Layout approval*	Project Engineer	send *Design* to drafter for final drawings [40]
39	Engineering manager	*Layout disapproval*	Project Engineer	rework layouts and resend *Parts Layout* message to Engineering manager for approval [37]
74	Liaison Engineer	*Request for advice*	Project Engineer	return *Advice* message [75]
78	Liaison Engineer	*Drawings*	Project Engineer	store for future reference
69			Quality Control	notice some discrepency between plane and plan send *Rejection Tag* message to Liaison Engineer [70]
68	Customer Inspector or FAA Inspector	*Customer pickup* or *FAA pickup*	Quality Control	determine if reported problem is valid send *Rejection Tag* message to Liaison Engineer [70]

80	Liaison Engineer	*Rework*	Quality Control	file and track disposition of problem
81	Liaison Engineer	*Accept parts*	Quality Control	file and track disposition of problem
86			Subcontractor	send a *Engineering Change Request* message to the Miscellaneous Changes Group [87]
66	Engineering Change Control	*Preliminary PRR*	Super Board	approve or reject entire change package and set priority send *Approved PRR* message to Engineering Change Control[67]
4	Customer Engineer or Engineering Change Control	*Change request work statement* or *PRR Tipsheet*	Test integration	determine what tests are necessary for change send *Test proposal* message to Designated Engineering Representatives [9]
28	Designated Engineering Representative	*Comment*	Test integration	possibly revise test plan and send *Test proposal* message to Designated Engineering Representative [9] send *Test plan* to Engineering cost and schedules [13] if the change is customer initiated, send *Test plan* message to Customer Engineer [62] otherwise, send a *Test Plan* message to Engineering Change Control [63]
7	Customer Engineer or Engineering Change Control	*Change request work statement* or *PRR Tipsheet*	Weights	determine effect of change on weight return *Weight* message [12] or [58]

6-2

INTENTIONAL MODEL FOR AIRPLANES, INC.

Sorts

```
change
change-proposal
change-request-document
class
crws (change request work statement)
drawing
engineer isa person
layout
manager isa person
person
test-plan
```

Functions

```
change(CRWS)
change(Drawing)
change(Layout)
class(Change)
cost(CRWS)
engineering-hours(CRWS)
manager(Person)
price(CRWS)
test-plan(CRWS)
```

```
weight(CRWS)
```

Relations

```
accepted-by(person, change-proposal)
approved-by(person, change-proposal)
approved-by(person, drawing)
approved-by(person, layout)
includes-features(airplane, change)
```

Individual models

Customer

```
∃Airplane, Change: want(customer, have(customer,
  Airplane)) ∧ includes-feature(Airplane, Change)
know(customer, can(customer-engineer, develop-change-
  proposal(customer-engineer, Change)))
can(customer, talk-to(customer, customer-engineer))
can(customer, accept-change-proposal(customer, Change-
  proposal))
know(customer, can(contracts, implement-change-
  proposal(contracts, Change-proposal))
```

develop-change-proposal(Performer, Change)

```
Add: ∃Change-proposal: have(Performer, Change-proposal)
     ∧ change(Change-proposal) = Change
```

accept-change-proposal(Performer, Change-proposal)

```
Add: accepted-by(Performer, Change-proposal)
```

implement-change-proposal(Performer, Change-proposal)

```
Pre: accepted-by(customer, Change-proposal)
Add: ∃Airplane: have(Performer, Airplane) ∧ includes-
     features(Airplane, change(Change-proposal))
```

Customer Engineer (ce)

```
know(ce, can(project-engineer, classify-change(project-
  engineer, Change)))
can(ce, lookup-crws(ce, Change))
can(ce, develop-crws(ce, Change))
know(ce, can(customer-engineering-manager, approve-
  change-request-document(customer-engineering-manager,
  change-request-document)))
can(contracts, offer-change(contracts, customer, CRWS))
know(ce, can(engineering-change-control, prepare-change-
  document(engineering-change-control, Change)))
know(ce, can(testing, create-test-plan(testing, CRWS)))
know(ce, can(weights, determine-weight(weights, CRWS)))
```

classify-change(Performer, Change)

```
Add: know(Performer, class(Change))
```

lookup-crws(Performer, Change)

```
Pre: class(Change) = "Standard Option"
Add: ∃CRWS: have(Performer, CRWS) ∧ change(CRWS) =
     Change
```

develop-crws(Performer, Change)

```
Add: ∃CRWS: have(Performer, CRWS) ∧ change(CRWS) =
     Change
```

approve-crws(Performer, CRWS)

```
Add: approved-by(Performer, CRWS)
```

prepare-change-request-document(Performer, CRWS)

```
Pre: know(Performer, weight(CRWS)) ∧ know(Performer,
     test-plan(CRWS))
Add: ∃Change-request-document: know(Performer, Change-
     request-document) ∧ change(Change-request-document)
     = change(CRWS)
```

```
approve-change-request-document(Performer, Change-
                request-document)
Add: approved-by(Performer, Change-request-document)

offer-change(Performer, Customer, Change-request-
                document)
Pre: appoved-by(customer-engineering-manager, Change-
     request-document)
Add: know(customer, Change-request-document)

create-test-plan(Performer, CRWS)
Add: know(Performer, test-plan(CRWS))
```

Test integration

```
can(testing, create-test-plan(testing, CRWS))
know(testing, can(der, approve-test-plan(der, Test-
  plan)))
know(testing, can(engineering-cost-and-schedules,
  determine-cost(engineering-cost-and-schedules, CRWS)))

create-test-plan(Performer, CRWS)
Add: know(Performer, test-plan(CRWS))

approve-test-plan(Performer, Test-plan)
Add: approved-by(Performer, Test-plan)

determine-cost(Performer, CRWS)
Pre: know(Performer, test-plan(CRWS)) ∧ know(Performer,
     CRWS)
Add: know(Performer, cost(CRWS))
```

Weights

```
can(weights, determine-weight(weights, CRWS))

determine-weight(Performer, CRWS)
Add: know(Performer, weight(CRWS))
```

Designated Engineering Representatives (der)

```
can(der, approve-tests(der, Tests))
```

approve-tests(Performer, Tests)

```
Add: approved-by(Performer, Tests)
```

Engineering Cost and Schedules (ecs)

```
can(ecs, determine-determine-engineering-hours(ecs,
  CRWS))
can(finance, determine-cost(finance, CRWS))
```

determine-engineering-hours(Performer, CRWS)

```
Pre: know(Performer, test-plan(CRWS)) ∧ know(Performer,
     CRWS)
Add: know(Performer, engineering-hours(CRWS))
```

determine-cost(Performer, CRWS)

```
Pre: know(Performer, test-plan(CRWS)) ∧ know(Performer,
     CRWS)
Add: know(Performer, cost(CRWS))
```

Division Finance

```
can(finance, determine-cost(finance, CRWS))
```

determine-cost(Performer, CRWS)

```
Pre: know(Performer, engineering-hours(CRWS)) ∧
     know(Performer, change(CRWS))
Add: know(Performer, cost(CRWS))
```

Pricing

```
can(pricing, determine-price(pricing, CRWS))
can(pricing, talk-to(pricing, contracts))
know(pricing, can(contracts, propose-change(contracts,
  customer, Change-proposal)))
```

determine-price(Performer, CRWS)
Pre: know(Performer, cost(CRWS))
Add: know(Performer, price(CRWS))

propose-change(Performer, Customer, Change-proposal)
Pre: know(Performer, price(Change-proposal))
Add: know(Customer, Change-proposal)

Contracts

can(contracts, talk-to(contracts, Customer))

change(Change) ⇒ know(customer, want(customer, Change))
  ∨ ~know(customer, want(customer, Change))

know(contracts, can(customer-engineer, implement-change-proposal(customer-engineer, Change-proposal))

implement-change-proposal(Performer, Change-proposal)
Pre: accepted-by(customer, Change-proposal)
Add:

Customer Engineering Manager (cem)

can(cem, approve-change-proposal(cem, Change-proposal)

approve-change-proposal(Performer, Change-proposal)
Add: approved-by(Performer, Change-proposal)

Project Engineer

can(project-engineer, classify-change(project-engineer, Change))

know(project-engineer, can(drafter, prepare-drawing(drafter, Layout)))

know(project-engineer, can(engineering-data-control, release-drawing(engineering-data-control, Drawing)))

know(project-engineer, plan(implement-change, release-drawing))

```
know(project-engineer, can(engineering-manager, approve-
   layout(engineering-manager, Layout)))
know(project-engineer, can(engineering-manager, approve-
   drawing(engineering-manager, Drawing)))
```

classify-change(Performer, Change)
```
Add: know(Performer, class(Change))
```

prepare-drawing(Performer, Layout)
```
Add: ∃Drawing: have(Performer, Drawing) ∧
     change(Drawing) = change(Layout)
```

release-drawing(Performer, Drawing)
```
Pre: approved-by(manager(Performer), Drawing)
```

approve-layout(Performer, Layout)
```
Add: approved-by(Performer, Layout)
```

approve-drawing(Performer, Drawing)
```
Add: approved-by(Performer, Drawing)
```

Engineering Manager (em)

```
can(em, approve-layout(em, Layout))
can(em, approve-drawing(em, Drawing))
```

approve-layout(Performer, Layout)
```
Add: approved-by(Performer, Layout)
```

approve-drawings(Performer, Drawing)
```
Add: approved-by(Performer, Drawing)
```

Drafter

```
can(drafter, prepare-drawing(drafter, Layout))
```

prepare-drawing(Performer, Layout)
```
Add: ∃Drawing: have(Performer, Drawing) ∧
     change(Drawing) = change(Layout)
```

Manufacturing Change Board (mcb)

```
can(mcb, decide-offerability(mcb, CRWS))
know(mcb, offerable(CRWS) ⇒ can(contracts, offer-
   change(contracts, customer, change(CRWS))
```

decide-offerability(Performer, CRWS)

```
Pre: ∃Change-action-memo: have(mcb, Change-action-memo)
     ∧ change(Change-action-memo) = change(CRWS)
Add: know(mcb, offerable(CRWS)) ∨ know(mcb,
     ~offerable(CRWS))
```

Engineering Change Control (ecc)

```
can(ecc, prepare-change-document(ecc, Change)))
```

prepare-change-request-document(Performer, CRWS)

```
Pre: know(Performer, weight(CRWS)) ∧ know(Performer,
     test-plan(CRWS))
Add: ∃Change-request-document: know(Performer, Change-
     request-document) ∧ change(Change-request-document)
     = change(CRWS)
```

Engineering Data Control (edc)

```
know(edc,
   equal(implement-change(edc, Change),
   concurrent(inform(edc, tooling, Change),
   inform(edc, materiel, Change),
   inform(edc, planning, Change),
   inform(edc, project-engineer, Change))))
```

Super Board (sb)

```
can(sb, approve-change-proposal(sb, Change-proposal))
```

approve-change-proposal(Performer, Change-proposal)

```
Add: approved-by(Performer, Change-proposal)
```

7

RECIPES FOR MULTI-AGENT ACTION

Although you will perform with different ingredients for different dishes, the same general processes are repeated over and over again. As you enlarge your repertoire, you will find that the seemingly endless babble of recipes begins to fall rather neatly into groups of theme and variations...

—Child, Bertholle and Beck, *Mastering the Art of French Cooking*

In the cases, I found that the people I studied did many things. I want to develop a concise way to say which things they did are the same, to describe what problems arise that require coordination and to identify where organizations have taken alternative approaches to similar problems. In short, I want to move from a simple description of the specific sites to a typology of the different kinds of coordination methods used.

1 A typology of coordination problems and methods

In my view, coordination problems are caused by interdependencies between various elements of a situation which constrain how particular tasks are performed. These problems require the actors to perform coordination methods to overcome the constraints. In this section, I will propose a typology of these coordination problems, based on the kinds of interdependencies that can arise.

To generate the typology, I group the four elements of my models—goals, actions, actors and objects—into two categories: (1) the objects that make up the world and in particular, the resources needed to perform actions (including the actors themselves); and (2) tasks, such as achieving a goal or performing an action. In any situation there may be multiple instances of each of these elements, possibly interdependent.

I simplify the typology by considering interdependencies only between pairs of items and focusing on the differences due to different kinds of items. Obviously, it is possible to describe situations where a group of items interact while no pair do. (For example, consider the following three tasks in a blocks world: place block A on B, block B on C and block C on A. Any pair of these tasks can be achieved, although the combination of the three cannot.) However, I argue that the kinds of coordination problems that arise in the general case are the same as those that develop between pairs (although the solutions to the general problem may be more difficult).

There are three possible pairs of these two elements (tasks and objects) taken two at a time ignoring order and therefore three kinds of interdependencies I consider: (1) task and task, (2) task and object and (3) object and object. I further divide task-task relationships into (1a) subtask relationships and (1b) overlaps (basically relationships up-and-down versus across the task hierarchy) because of differences in the kinds of coordination methods used. This typology is shown in Table 7.1.

Each kind of interdependencies gives rise to a set of coordination problems; there are multiple coordination methods that can be used to address each kind of problem. The typology groups coordination methods together by the problem they address.

Table 7.1. *Typology of dependencies between tasks and objects.*

<table>
<tr><th></th><th>Tasks</th><th>Objects</th></tr>
<tr><td rowspan="2">Tasks</td><td>1a Subtask</td><td rowspan="2">2 Task or resource assignment</td></tr>
<tr><td>1b Task overlaps</td></tr>
<tr><td>Objects</td><td></td><td>3 Object interdependencies</td></tr>
</table>

Of course, this grouping is very coarse. For example, there are important differences between different kinds of objects and different ways tasks can be interdependent. I therefore break up some of the cells into finer distinctions; these distinctions are discussed as I present the typology in more detail.

Table 7.2 presents a summary of the coordination problems identified and the section of this chapter in which they are discussed; Table 7.3 summarizes all

Table 7.2. *Typology of coordination needs and corresponding chapter section.*

Coordination need	*Section*
Type 1a: task-subtask interdependencies	3
task decomposition	3.1
goal induction	3.2
Type 1b: other task interdependencies	4
eliminate or reduce interdependency	4.2
two tasks create same object	4.3
tasks are duplicates	4.3.1
tasks specify different aspects of the object	4.3.2
object created by one task is used by another	4.4
order tasks	
transfer object from one task to another	
two tasks use the same object	4.5
uses of object must be scheduled	
uses of object conflict	
Type 2: task-object interdependencies	5
identifying what objects are needed by the task	5.1
identifying what objects are available	5.2
choosing particular set of objects	5.3
in the case of an actor, getting the actor to work on the task	5.4
Type 3: object interdependencies	6
trace effects of tasks	6.1

recipes introduced in this chapter.

In the remainder of this section I will briefly describe the two elements, tasks and objects, and explain why I grouped the elements of my models in this way.

Table 7.3. *Coordination recipes.*

Coordination method	*Recipe*
Type 1a: task-subtask interdependencies	
Task decomposition	`can(actor, plan(actor, Goal))`
Integration	`can(actor, integrate(actor, Subtasks))`
Supertask induction	`can(actor, induce-supergoal(actor, Subtasks))`
Type 1b: other task interdependencies	
Create-create	

Merging duplicate tasks	`know(actor, task`$_1$`)` $\wedge$ `know(actor, task`$_2$`)`
	`can(actor, check-for-duplicate-tasks(actor, Task`$_1$`, Task`$_2$`)`
	`can(actor, merge-tasks(actor, Task`$_1$`, Task`$_2$`))`

Reusing existing results	`know(actor, subtask`$_1$`)` $\wedge$ `know(actor, subtask`$_2$`).`
	`can(actor, check-for-duplicate-task(actor, Task`$_1$`, Task`$_2$`)`
	$\forall$`Task` $\in$ `known-tasks: have(actor, result(Task))`
	or
	`can(actor, lookup-result(actor, Task))`

Create-use	
User requests object	request(actor$_1$, actor$_2$, informref(actor$_2$, actor$_1$, λX: X=result(task)))
Creator knows user needs object	know(creator, next-task-in-plan(plan, current-task)) ∧ know(creator, task-assignment(next-task-in-plan(plan, current-task), user)
Third party acts as clearinghouse	some combination of above methods
User searches for object	can(user, search-for-result(user, Task))
Use-use	
Type 2: task-object interdependencies	
Identifying task requirements	can(assigner, calculate-task-requirements(assigner, Task))
Identifying resources	
Single resource	know(assigner, can(performer, task-requirements(performer, task)))
Calculate appropriate resource	can(assigner, calculate-which-performer(assigner, Task))
Ask informer to name performer	know(assigner, ∃Actor: know(informer, can(Actor, task))) ∴ request(assigner, informer, informref(informer, assigner, λX: can(X, task)))

Market-like assignment	`know(assigner, ∃X ∈ actors: can(X,task))`
	`∴ ∀X ∈ actors:`
	`request(assigner, X, informref(X, assigner, λY: Y = ability(X, task))`

Bulletin board	`know(assigner, ∃X ∈ Actors: can(X,task))` `∴ ∀X ∈ Actors: request(assigner, X, task)`
Choosing particular resources	`can(assigner, pick-best-resource(assigner, Resources))`
Acquiring actor's effort	`request(assigner, performer, task)`
Assigning resources	`can(assigner, check-if-busy(assigner, Resource))`
Assigning task assignment task	`know(assigner, ∃Actor know(informer, can(Actor, task)))` `∴ request(assigner, informer, task)`
Type 3: object interdependencies	
Checking for object interdependencies	`can(actor, check-model-for-interdependencies(actor, Model, Component))`
Evaluating interdependencies	know(actor, plausible(want(actor, change($component_1$)) ∧ physically-interdependent($component_1$, $component_2$), want(actor, change($component_2$))))

1.1 Objects

I consider everything used or affected by actions together in this category. Objects that are not somehow used or affected by an action are not considered; if they are not involved in the actions of some actor, then they are irrelevant to the analysis of the behaviour of that actor. For example, in the case of the car company, the objects include tools, raw materials, parts, partially completed assemblies, information such as designs or process instructions and the efforts of the employees of the company.

Note that actors are viewed simply as a particularly important kind of resource. I grouped actors and other kinds of resources together in this way because I wanted to explicitly consider the question of matching actors and tasks and the issues that arise in assigning tasks to actors parallel those involved in assigning resources to tasks.

Of course, there are other important differences between objects. I consider in particular two dimensions: shareablity and reusability, as shown in Table 7.4. The first dimension describes how many actions can use an object at a single time. Shareable objects, such as information, can be used by multiple actions simultaneously. Most other objects, such as effort, raw materials or tools, are non-shareable, since only a single action can be using them at one time. Formally, if we have a predicate,

```
use(object, action, situation),
```

indicating that the object is used by the action in the given situation, then

Table 7.4. *Examples of objects classified by shareability and reusability.*

	Shareable	*Non-shareable*
Reusable	Information	Tools
Consumable		Raw materials

$$\begin{gathered}\texttt{use(object, action}_1\texttt{, situation}_1\texttt{)} \wedge \\ \texttt{use(object, action}_2\texttt{, situation}_2\texttt{)} \wedge \\ \texttt{action}_1 \neq \texttt{action}_2\end{gathered}$$

is valid for all $\texttt{situation}_1$ and $\texttt{situation}_2$ for shareable objects, while for the nonshareable objects,

$$\begin{gathered}\texttt{use(object, action}_1\texttt{, situation}_1\texttt{)} \wedge \\ \texttt{use(object, action}_2\texttt{, situation}_2\texttt{)} \wedge \\ \texttt{action}_1 \neq \texttt{action}_2 \Rightarrow \\ \texttt{\~{}overlap(situation}_1\texttt{, situation}_2\texttt{)}.\end{gathered}$$

The second dimension, consumable/reusable, pertains to use at different points in time. Consumable objects, such as raw materials, can only be used once; use by one action prevents any other action from using that object at any other time. Reusable objects—things like tools or information—can be used and reused at different times. It appears that all shareable objects are also reusable, that is, that there seem to be no objects that can be used by multiple actions simultaneously but are consumed in the process. Formally,

$$\begin{gathered}\texttt{use(resource, action}_1\texttt{, situation}_1\texttt{)} \Rightarrow \\ \texttt{\~{}}\exists\texttt{Action}_2\texttt{, Situation}_2\texttt{:} \\ \texttt{use(resource, Action}_2\texttt{, Situation}_2\texttt{)}\end{gathered}$$

1.2 Tasks

This category includes both achieving goals and performing actions. Goals and actions are usually considered as quite distinct. Both describe an outcome state, but actions have several other characteristics. First, actions usually have preconditions, that is, an action may only be applicable in a particular state of the world. For example, an action may require as a precondition that an object be present or that it have a particular attribute. In the cases I studied, knowledge is an important kind of precondition since an actor may need to have some knowledge before it can execute an action. Second, actions may have unintended side-effects which are not part of the goal.

However, I believe that for understanding the coordination issues that arise in the organizations I studied it makes more sense to consider actions and goals together. I call both of them *tasks;* a task can be achieving some goal or to performing some action. This view makes clear the parallels between decomposing goals into subgoals and decomposing them into actions. By treating higher-level goals as actions to be accomplished by the subunit, this view allows us to treat assignment of goals to a subunit in the same way that we consider assigning actions to individuals. Both goals and actions are descriptions of the task to be undertaken by the particular subunit to which it is assigned.

2 Recipes

As defined in Chapter 1, a recipe is "what someone knows when they know how to do something." I represent "what someone knows" in an augmented first-order logic, as discussed in Chapter 2. Therefore, a recipe is a set of formulas that describes what an actor needs to know or descriptions of actions that it must be able to perform in order to perform a coordination method. Some of the knowledge can be described generically; in other cases, it is domain-specific and can only be described in general terms.

Note that this definition is not the same as the kitchen definition of a recipe as a sequence of steps to be taken. Rather, it is more analogous to the knowledge an expert chief would have to allow him or her to generate the sequence of steps.

In the following sections, I describe the kinds of coordination problems that arise in the categories developed above. For each of the coordination needs, I present the recipe for various coordination methods that can be used to address it. I first give examples of the coordination method drawn from the case studies. These examples are set off by being printed in italics. I then provide a general formalization of the recipe. These recipes are distinguished by being printed in a distinctive font (`courier`) and set in a box.

When an actor uses a recipe to carry out a coordination method, it usually results in some communication activity. I periodically illustrate the use of the recipes by showing how actor deduces what to do and the set of messages that it sends as a result.

2.1 Sources of recipes

The primary source for the recipes is the case studies discussed in the preceding three chapters. In this study, I focus on the kinds of coordination problems that arose in engineering organizations. Other kinds of organizations may have somewhat different kinds of problems, although there is likely to be substantial overlap. I believe that these other as yet unstudied coordination tasks should fit into the typology presented here, in as much as they can be analyzed along the same dimensions, but (perhaps) with different values for dimensions than any of the tasks I studied.

Furthermore, because I studied the organizations at one point in time, I mostly saw short-term coordination processes and not more long-term ones. For example, decisions about how to decompose a goal and design individual actions are taken only rarely, when the organization is designed.

In some cases, it is clear that there are other possible ways of performing the coordination tasks and I have attempted to discuss these ways. One useful method for generating these alternatives is to consider possible distributions of the coordination knowledge identified. Another variation is between knowing something and being able to calculate it. In some cases the actors may perform a particular set of actions because they are following a script. For example, an action might be coordinated in advance by the creation of a plan which is then executed by the members of the organization. In these cases, the analysis of the necessary knowledge is still applicable, but the knowledge in question may be held by the actor who creates the plan.

3 Category 1a: Dependencies between tasks and subtasks

The first category of coordination methods are those that manage the interdependencies between tasks and subtasks. Tasks and subtasks interact because performing the subtasks is necessary to perform the tasks.

3.1 Task decomposition

One way to manage this dependency is to decompose the task into subtasks to be performed by individual actors. This is the basically the planning process, briefly discussed in Chapter 2. To perform this decomposition, an actor must know what tasks are to be achieved and know (or be able to generate) possible subtasks (or primitive actions) and their preconditions and effects.

If the subtasks are known in advance by some actors, those actors can create a plan for performing the task. The plan may then be distributed to other actors to be carried out. Otherwise, a more adaptive process is required to recognize and decompose tasks on the fly. In practice, organizations do some of both. For example, one actor may create an abstract plan and distributing high-level subtasks of the plan to other actors to carry out.

In Car Co., the process engineers decompose the design into specific operations the assembly workers can perform to assemble the cars. The process engineers get the design of the car from the design engineers. In addition, they know what the workers on the assembly line can do in the time available, what tools are available or can be built, etc.

In Car Co., the production scheduling group decides where to build cars and even the order in which they should appear on the assembly line. They know how many of each car is needed from the marketing department. They also know the constraints of the assembly line, such as how many of each model can be manufactured.

In Airplanes, Inc., a planning group determines what steps are required to implement a change, including design, manufacturing parts, etc. This group knows what steps are required to implement a change. In addition, they know how long each operation should take to perform, allowing them to develop a schedule for the plan.

As discussed in Chapter 2, I assume that actors have a plan operator with which they can generate a plan to achieve their goals. The recipe for this case is simply some kind of action that generates a plan:

```
plan(Performer, Goal)
Preconditions: know(Performer, Actions)
Effects:       know(Performer, Plan) ∧ achieves-
               goal(Plan, Goal)
```

Depending on the situation, more domain-specific knowledge may be used, but this recipe is all that can be said in general.

3.1.1 Task integration

If multiple subtasks are performed, it may be necessary to integrate their results. This integration step is frequently viewed as a kind of coordination task. However, I believe that integration can be viewed simply as another part of performing the task, that is, the task is decomposed into multiple subtasks, one of which is to integrate the results.

To do this integration requires knowing the results of the actions. The integration subtask is thus dependent on the other subtasks, as discussed below. In addition, the actor must have some domain-dependent knowledge about how to do the integration.

In Computer Systems Co., the integration group can integrate changes to various modules into the final system. This group knows how to recompile and link the different modules into a working operating system and how to test the resulting system.

Integration may be represented by an operator such as:

```
integrate(Performer, Tasks, Subtasks)
Preconditions: have(Performer, results(Subtasks))
Adds:          have(Performer, results(Tasks))
```

The exact axiomatization depends on nature of the results to be integrated. For example, if the results are parts to be assembled, then the precondition is to actually possess the parts; if the results are knowledge, then the actor simply needs to know the results.

3.2 Supertask induction

An alternative approach to managing the dependency between tasks and subtasks is to determine what primitive subtasks are possible and choose supertasks that can be achieved with these actions. To do this induction requires knowing what primitive subtasks are possible, their preconditions and effects and being able to generate possible supertasks that could be achieved. It may also be necessary to have some way to choose between alternative possible supertasks.

I represent this induction by a domain specific action, `induce-supertask`. This action gives the actor a new goal, one that can be achieved with the given subtasks.

```
induce-supertask(Performer, Subtasks)
Adds:           want(Performer, Task) ∧
                can-achieve(Performer, Task)
```

I did not see any examples of this in the cases, but I point it out here because it seems logically consistent with this framework.

4 Category 1b: Dependencies between different tasks

The second category of coordination methods are those that manage the interdependencies between tasks other than subtask relationships. Obviously, there are many possible interdependencies between tasks; I therefore divide this category more finely. Many researchers have tried to categorize the kinds of interdependencies that arise between pairs of goals or actions; some of these efforts are summarized at the end of this chapter.

I believe that it is most productive to view tasks as interacting through their effects on a common set of objects in the world. This conception is similar to Star's (1989) notion of boundary objects. Star introduces boundary objects as a channel for actors with different goals and viewpoints to cooperate in solving heterogeneous problems. In her words, boundary objects are "plastic enough to adapt to local needs... yet robust enough to maintain a common identity across sites" (p. 46). Star describes different kinds of boundary objects she found in her study of a natural history museum, but she notes that her list was not exhaustive and suggests looking for other kinds of boundary objects.

To organize this category of coordination problems, I therefore develop a typology of the ways different tasks interact, by considering the ways they interact with objects in the world. I propose a simple typology of possible dependencies between two tasks based on what kind of common object is used (shareable/reusable) and how it is used by the two tasks.

One useful distinction Star (1989) does not make explicitly is the fashion in which tasks use the common objects. I consider only two basic operations a task can perform on an object: creation and use. An object is used if it appears in the precondition list of some action; it is created if it appears in the effect list. (For consumable objects, use is consumption.) For two tasks using a common object, there are three possible combinations of these two modes of use: create-create, create-use and use-use.

Table 7.5 shows the dependencies that arise between tasks for these three combinations of create and use and the three different kinds of objects described above. Temporal relations obviously mediate many of these dependencies; for some kinds of objects, a constraint may exists only if the tasks use the objects at the same time. These interdependencies may not be visible at higher levels of abstraction of the tasks; they may become apparent only when the goal is decomposed into a set of specific actions.

Table 7.5. *Typology of coordination problems by operations and type of common object.*

use / *type of object*	*create-create*	*create-use*	*use-use*
shareable	eliminate duplicate negotiate object	order tasks transfer object	no conflict
non-shareable/ reusable	share object	order tasks transfer object	schedule use
non-shareable/ consumable	economies of scale	order tasks transfer object	conflicting goals

Reading across the columns, there are three primary classes of coordination problems: create-create, create-use and use-use.

Create-create. If an object is created by two tasks, then it may be possible to merge the two tasks or partially or completely eliminate one of them. Alternately, the two tasks may specify different aspects of the object, possibly requiring negotiation to arrive at a consensus.

Create-use. If an object is created by one task and used by another, then there is a temporal dependency between the two, requiring that the tasks be performed in the correct order and that the common object be transferred from one to the other. (This problem is similar to the "producer/consumer" problem in computer science.) This relationship frequently holds between steps in a process. For example, in assembling a car, the spot welding operation takes components as input and creates a body; painting takes body is taken as input and creates painted body.

Use-use. If two tasks both use an object, and the object is shareable, then there is no conflict. If the object is not shareable but is reusable then the use of the object must be scheduled. This implies that one task must be chosen to be performed first and the other task made to wait. If the object is not reusable, then the two tasks can not both be performed with the available resources. Additional

resources must be acquired or one of the tasks dropped. (These final two problems are related to the "mutual exclusion" problem in computer science.)

A single task can perform both kinds of access on a single object; for example, modification of an object can be modelled as use followed by recreation. For non-shareable objects, the resulting dependencies are the combination of the dependencies from the individual operations.

4.1 Avoiding constraints

One approach that may be useful for managing any kind of dependency is to avoid or eliminate it. Since dependencies, in my view, arise when two tasks both use some common object, one strategy for eliminating dependencies is to restrict access to the common object to a single task.

A variant of this strategy is to internalize the need to coordinate in a single actor, by having one actor perform all actions that access a particular common object. This strategy does not eliminate the dependency between the tasks nor the need to manage them, but it does permit the use of much simpler management techniques as only a single actor is involved. This actor can, for example, decide which operations to perform, without having to check if these operations conflict with someone else. This centralization seems to be a common reason for various kinds of role specialization. Of course, centralizing access does not completely eliminate the need for coordination between actors because the objects themselves can be interdependent, as discussed below, but it does eliminate one class of dependencies.

In all of the sites studied, the effects of potential dependencies between several changes to a single part are mitigated by having a single engineer make all changes to the part. For example, in Computer Systems Co., having programmers specialize in a particular module eliminates the need to manage access to modules, since only a single programmer can make changes.

4.2 Conflicting creation of objects

The first category of task interdependencies I will consider are conflicting creates, that is, the two tasks both create the same object (column 1 of Table 7.5). There are two subcases to consider. First, an identical object may be created by the two tasks or alternately, the two tasks may be duplicates. For example, two engineers may try to design the same part. Second, the two tasks may specify different aspects of an object, thus requiring cooperation between the tasks. For example, the interface between two modules of an operating system may be developed and used jointly; the specification of a customer's airplane is negotiated between the airplane company and the customer.

4.2.1 Tasks are duplicates

If the two tasks create the same object, then it may be possible to eliminate one of the tasks, thus reducing the total effort required. For example, having two engineers both design the same part is usually a waste of effort. Even if the resulting object is not shareable (e.g., if the task is to make a new part), there may still be economies of scale that make it cheaper for one actor to do the task twice than for two actors to do the task once, again suggesting merging the tasks.

To compare the tasks requires first that some actor know both tasks. Knowing both tasks can be represented by the following piece of a recipe:

$\text{know(actor, task}_1\text{)} \wedge \text{know(actor, task}_2\text{)}.$

Second, the actor must be able to tell that the tasks are the same. I assume that the actor can calculate a predicate `same-task` that is true for equal tasks (that is, tasks that create the same output):

check-for-duplicate-tasks(Performer, Task_1, Task_2)	
Adds:	know(Performer, same-tasks(Task_1, Task_2)) ∨ know(Performer, ~same-task(Task_1, Task_2))

Checking for duplicates before tasks are performed. If the check for the duplicates is done before either task is performed, then it is possible to merge the two tasks together and perform only one of them. How the tasks are merged depends on the context. For example, when decomposing tasks into subtasks, an actor may simply avoid creating duplicate subtasks. If the tasks have already been decomposed and assigned to actors, then the group may have to decide which tasks not to do.

Checking for duplicates after one task has been performed. If the tasks are performed at different times, the actor performing the later task may be able to simply reuse the result of the earlier task, if it can find it. To do this, there must be some way to find the results of earlier tasks, to tell that the current and past tasks are the same and to use the existing results found.

This cell of the typology is where I classify the coordination method performed by response center described in the first example in the introduction.

> *In Computer Systems Co., the response center and marketing engineers check if a reported problem is already in a database of known problems. If it is, then the solution to that problem report can be used.*

One possibility is that the actor may simply have the result for some set of tasks. The recipe for this situation looks like this:

```
∀Task ∈ known-tasks: have(actor, result(Task))
```

Alternately, the actor may have some action it can take to find the previously found result, such as:

```
lookup-result(Performer, Task)
Preconditions: Task ∈ known-tasks
Adds:          have(Performer, result(Task))
```

Presumably this action costs less than the other actions that can be used to find the result, so this action is tried first.

4.2.2 *Duplication of tasks for redundancy*

It may be that the tasks are duplicated to provide redundancy (Rochlin, et al., 1987). For example, several sensors might measure the temperature of a reactor to protect against the failure of any single sensor.

For shareable resources such as information, there must be an additional task to ensure that the output of the redundant tasks agree. This task might consist, for example, of each user of the object monitoring all outputs and taking some recovery action if they disagree. Each such task is thus a user of the outputs; the associated coordination problems are discussed in section 4.3.

For nonshareable resources, each task independently creates some amount of the resource, thus protecting against the failure of any single process. In this case, tasks using the resource may have a choice of which input to use; the associated coordination problems are discussed in section 5.3.

4.2.3 *Tasks create different aspects of the object*

The second possibility is that the two tasks have different perspectives on the objects being created. For example, in a design problem, the interface between two components may be explicitly negotiated by the actors designing the components. This process can be modelled as an exchange of proposals until a mutually acceptable object is found. Negotiating the object requires some domain specific knowledge about the tradeoffs between various values of the object's features and some way to evaluate a particular set of features.

> *In Computer Systems Co., the interface between two modules may be negotiated by the engineers developing the two modules.*
>
> *In Airplanes, Inc., the airplane configuration is used to ensure that the customer and the company agree on the desired features of the airplane and by all engineers to ensure they are designing the same aircraft. In this case, the common object is defined by negotiation between the company's customer engineers and the customer.*

4.3 Output of one task is input of another

A particularly common case is that the common object between two tasks is created by one task and used as an input by another. In this case, there is a temporal dependencies between the tasks. Coordination is necessary to ensure that the tasks are performed in the correct order and to manage the transfer of the object from one task to the next. As mentioned above, the output of many tasks may be the input to an integrating task.

> *In Car Co. and Airplanes, Inc., designs are created by a design task and used by a parts manufacturing task. The parts manufacturing task in turn creates parts which are used by a product assembly task.*
>
> *In all sites, the individual components of the products are created separately and assembled into the final product.*

To ensure that the tasks are performed in correct order requires that the actor performing the first task know which actor should get the object next and the second actor knowing that it should wait for the result of the first.

There are several possible solutions to the first problem. First, the actor may simply know that the result is to be sent to a particular actor.

> *In Computer Systems Co., the distribution group knows how to find the customers who need the latest release of the system. The group must know which other actors are expecting the result.*

This knowledge is probably best represented as part of the plan executed by the first actor; the final step is an action to tell or give the second actor the result, such as:

```
inform(creator, user, result(task))
```

The plan may include this final step because the user asked the creator to perform the task and to return the result.

In Computer System Co., customers call the response center, describe the problem they want fixed and (implicitly or explicitly) request that they be sent the solution.

In this case, the creator will have a goal of sending the result to the second actor as a result of a request such as:

```
request(user, creator, informref(creator,
      user, λX: X = result(task)))
```

Second, the first actor may know the goal structure and thus what the next task is and which actor performs it by virtue of a recipe such as:

```
know(creator,
  next-task-in-plan(plan, current-task)) ∧
  know(creator, task-assignment(next-task-in-
  plan(plan, current-task), user)
```

In this case, the creator can reason from the knowledge preconditions of the next task that the user needs the result of the current-task and therefore transfer the result.

Finally, a third actor may oversee the process, acting as a clearinghouse for the results.

In Airplanes, Inc., the engineering change control group shepherds changes through the change process, tracking their status and generally ensuring that things are happening.

In Computer Systems Co., a document library centralizes communication of changed documents. The group knows which other actors are expecting the result of changing a document.

In this case, the first actor needs to know to send results to the central actor; the central actor must know which actor needs the result next. The execution of the applicable recipes results in the following communication:

```
inform(creator, clearinghouse, result(task))
 inform(clearinghouse, user, result(task))
```

The second problem, ensuring that the second actor waits for the result of the action performed by the first actor, can be modelled by assuming that the result is a precondition for the action of the second. Therefore, the second actor will not be able to perform the action until it gets the result. To get the result, the second actor can either wait for the first actor to send it, as described above, or actively seek it out.

The seeking out would be represented by an action similar to the lookup-result action discussed above.

```
search-for-result(Performer, Task)
Preconditions: Task ∈ known-tasks
Adds:          have(Performer, result(Task))
```

In this case, the first actor need only put the result of the first action somewhere where the second actor will be able to find it.

4.3.1 Synchronizations

Two tasks may have to happen at the same time; furthermore, they should either both happen or both not happen. For example, for two movers to lift a piano requires that both lift their ends at the same time.

In all sites, two changes may need to be implemented together, since each depends on a change implemented by the other. Such changes indicate the changes on which they depend; neither change can be implemented separately.

For example, in Car Co., when the supports for the rear seats were changed to integrate a spacer for the trunk lining, the separate spacers being used should have been eliminated.

One way to model this action is to assume that the output of each task is an input or precondition of the other. Therefore, the tasks must temporally precede each other, or alternatively, they must be executed concurrently.

Synchronizing the execution of the tasks is difficult. The actors performing the tasks must both know that they need to be synchronized. If they

both know both tasks they can figure this out by reasoning from the preconditions and effects of the tasks.

If each actor knows the other actor performing the task, they can pick a time at which to perform the two tasks using some group decision process.

Alternately, a third actor may signal both actors to proceed at the same time.

4.3.2 *The user constrains the creator of the objects*

So far we have assumed that one task creates an object which is then independently used by some second task. It is possible that the task using the object may constrain how the object is created.

> *In Computer Systems Co., the user creates and the response centre uses the customer complaint. The two may negotiate the details of the complaint, for example, by iterating the process (i.e., the customer files a complaint, the response centre asks for more details, the customer supplies them, etc.) or in a continuous dialogue.*

One way to model this process is to split the process of creating and using the object into two actions. The resource is first jointly created by the two actors and then used as input by one of them for a second action.

Alternately, some of the knowledge about the constraints of either task can be moved from one actor to another. This final case has three interesting subcases. First, some of the customer's knowledge can be transferred to the response centre, for example, by having the response centre recreate the situation on their own computers. Second, some of the response centre's knowledge can be made available to the customer. For example, at the site studied, a computer system had recently been developed that allowed customers to do their own searches through the database of known problems. Finally, a third party may have some of both actors' knowledge and be able to mediate between them. In some cases, for example, the customer engineer responsible for the site may investigate the problem and file a change request.

4.4 Conflicting use of objects

When two tasks both require some common object as input, they have a resource constraint. Resource constraints are not inherent in the nature of the tasks but are rather a product of the way the tasks have been assigned to actors or other resources.

4.4.1 *Shareable objects*

If the object is shareable, then there is no constraint; each action may simply use the object as it requires. One way to avoid coordination problems with shareable objects therefore is to freeze the common object, that is, to allow actions to only use the object. Which common objects are frozen depends on organizational criteria. For example, a common object used by tasks carried out by two actors who do not communicate (e.g., actors in different groups within the organization) is likely to be frozen to eliminate the need for those two actors to coordinate.

In Computer Systems Co., the interface the system presents to the end user is not changed in order to eliminate the need to coordinate the end user's use with the development of the system.

4.4.2 *Non-shareable, reusable objects*

If the resource, like a tool, is not shareable but is reusable, then:

$$\begin{array}{c} \mathtt{use(object,\ action_1,\ situation_1)} \wedge \\ \mathtt{use(object,\ action_2,\ situation_2)} \wedge \\ \mathtt{action_1} \neq \mathtt{action_2} \Rightarrow \\ \mathtt{\sim overlap(situation_1,\ situation_2)}. \end{array}$$

Therefore, to avoid this constraint, the actors must either choose different objects or use the same object at different times. Choosing different resources will be considered in the next section.

A special case of this constraint occurs when a single actor has more than one task to be performed, in which case the actor's time is the limited resource.

In Computer Systems Co., software engineers have to pick which problem reports to investigate first.

To ensure that the times during which the object is used do not overlap basically involves assigning the object to be used exclusively by a particular action or actor. Assignment mechanisms are discussed in the next section.

In the reorganized Computer Systems Co., engineers check code out of a library to prevent simultaneous updates.

4.4.3 *Non-shareable, consumable objects*

If the object is consumable, then the two tasks can not be simultaneously performed. I assume that the tasks have been described at an atomic level, so reducing the amount of the resource needed is not an option, or at least, is interpretable as not performing some task.

There are several alternatives. First, one of the tasks can be abandoned. One approach is first-come, first-served; which ever task gets the resources simply uses them and the other action does not.

Second, some actor may know the two tasks and evaluate the tradeoffs between them. Picking the order for the tasks or choosing which should be done or not done can be done randomly or based on the particular actor's preferences. It may be preferable that this choice reflect organizational priorities.

Third, more of the necessary resource can be assigned to the task, using mechanisms discussed in the next section.

5 Category 2: Dependencies between tasks and objects

The second category of coordination tasks involve assigning objects and especially the effort of actors, to tasks. As discussed above, I view the efforts of actors as a special kind of resource. In this section I mostly discuss the problem

of assigning a task to a particular actor to be performed, but the techniques discussed can be use to assign any kind of resource to a task.

I classify the second example mentioned in the introduction, the routing of problem reports to a software engineer capable of fixing them, in this broad category. In the most general case, this assignment can involve hiring or firing employees, acquiring new tools or raw materials, etc. This assignment may be done in a hierarchical fashion as subunits of the organization are assigned to work on high-level goals.

Typically, some actor identifies a task that needs to be performed and assigns another actor and the necessary resources to the task. I call the actor who knows the task the *assigner*; the actor who eventually performs the task, the *performer*. In order to assign resources to a task, the following steps must be performed:

1) identifying what resources are required by the task;
2) identifying what resources are available;
3) choosing a particular set of resources;
4) in the case of an actor, getting the actor to work on the task.

In my analysis, I consider these steps in this order. Obviously these steps are interdependent. For example, the way the requirements of the task are characterized depends on what kind of actors are available.

In principle, these steps can be performed in any order; for example, tasks can be chosen that can be performed with objects available (and that achieve higher level goals). One manager interviewed suggests that in software development, a manager may divide the project into modules based on abilities of programmers. I did not, however, observe this order of steps for the engineering change processes that I studied.

5.1 Identifying task requirements

The first step in the task assignment process is to determine what type of resources the task requires. Identifying task requirements requires domain-specific knowledge about the task, but the knowledge may be quite abstract.

In Computer Systems Co., in order to pick a software engineering group to fix a problem, a marketing engineer needs to know that the particular problem seems to be in the file system (for example) and that a particular group is responsible for the file system, but not anything about how to fix any kind of problems.

I assume that actors have an action that calculates what kind of actor (or other resource) is required by a task:

```
calculate-task-requirements(Performer, Task)
Effects:        know(Performer, task-
                requirements(Task))
```

This level of indirection—that is, having the assigner know which performers can manage the task requirements instead of knowing directly about tasks—avoids having the assigner know something about all possible tasks. We then assume that all assigners know:

```
know(assigner, can(performer, task-
  requirements(Task)) ⇒ can(performer(Task)))
```

The assigner may need to know what kind of actors are available to be able to characterize the task requirements along the same dimension as the actors are differentiated. Many researchers have considered the question of differentiation between actors. There are two extremes: specialists and generalists. In a specialist model, only one actor can perform any given task. In the generalist case, any actor can. In reality, things are rarely so precise, but organizations will be located along the spectrum between these two extremes. It may be possible to design systems which flexibly move from specialists to generalists under varying conditions.

In Computer Systems Co., programmers are specialists; to assign a problem report to an actor requires determining what part of the system is involved and assigning the problem to the appropriate actor. Other divisions of the same company use generalists actors; an incoming bug report is simply assigned to the next available actor to be fixed. (These two bases for organizing software maintenance are sometimes called module ownership and change ownership (Embry and Keenan, 1983).)

5.2 Identifying actors who can perform the task

If the actor who knows the task has a goal of having the task performed but can not (or will not) perform the task itself, then it must identify which other actors can perform the task.

5.2.1 Assigner knows a potential performer

The simplest possibility is that the assigner might know which actors can perform the task from the task requirements. Note that there may be many possible performers, as in the case of a generalist system, or only one.

For example, there may be a single actor that performs all tasks of a particular kind; each actor simply knows the association from a particular type of task to the performer.

In Car Co. and Airplanes, Inc., engineers rarely do the detailed design of components themselves. Instead, this task is performed by a designer who works with the engineer. When the engineer needs a detailed design worked out, he or she simply calls the designer.

In Car Co. and Airplanes, Inc., the drafting for all final drawings is done by workers in a drafting room. When an engineer or a designer need a final drawing prepared, they contact the drafting room to have it done.

In Computer Systems Co., customers only need to contact the response center to have a problem fixed.

This can be represented as the assigner knowing:

```
can(performer, task-requirements(task))
```

for various classes of task-requirements.

An assigner might remember a performer found earlier for a similar task, especially if the particular task assignment is done repeatedly or if searching for an actor is expensive compared to the perceived benefit of finding an alternative performer. As with any learning, there is a danger of over-generalization: an assigner may send tasks to the wrong performer because he knows him.

In Computer Systems Co., a marketing engineer can learn which software engineer is responsible for a particular part of the system and communicate directly with that engineer.

In Car Co. and Airplanes, Inc., when a part is redesigned, rather than attempting to find a new supplier to manufacture the new part the contract of the current supplier is often extended.

Again, this can be represented by which actors the assigner knows can perform the task.

In other cases, the assigner may be able to calculate from some characteristic of the task and knowledge about the organization which actors are potential performers.

In all sites, if a downstream group encounters a problem while processing a new component, they contact the engineer whose name is on the drawing to have the problem resolved.

In Car Co., the material scheduling group tells suppliers when to ship parts, how many to ship and where to ship them. This group is given the production schedule which indicates how many of each model of car are to be manufactured and when the cars will appear on the assembly line. They also know which parts are required for each car and where they are used on the assembly line.

In all sites, problem reports are routed to the engineers responsible for the affected parts. To do this routing, an actor must know what components are affected and which engineers are responsible for those components.

In Computer Systems Co., this knowledge is distributed hierarchically: the response center determines which product is involved and therefore which marketing engineer, the marketing engineer determines which system and which project and the project contact, which specific module and which engineer.

This can be represented by assuming the actor has an operator that moves directly from the representation of the task to knowledge about which actor can perform it:

```
calculate-which-performer(Performer, Task)
Adds:          ∃Actor: know(Performer, can(Actor,
               Task))
```

5.2.2 Assigner asks someone who knows

A second way to find potential performers is to ask other actors, *informants*, for the information. Morgenstern (1988) discusses a similar problem, that of an actor planning an action that requires the use of the knowledge of another actor. To do this, the assigner must know which actors are likely informants and be able to communicate with them. Note again that there may be multiple informants and potential performers. The informant role may be filled by various forms of technology, such as a yellow pages, computerized or on paper.

There are two possibilities. First, the informant may tell the assigner how to find a performer.

> *In Car Co., one engineer received a call asking him about some problem in a part for which he was no longer responsible. He therefore told the caller how to contact the correct engineer.*

We can model this situation as follows: the assigner has the following recipe:

```
want(assigner, task)

know(assigner, ∃Actor:
   know(informer, can(Actor, task)))
```

Therefore, in order to accomplish its goal, the assigner performs the following communication action:

```
request(assigner, informer, informref(informer,
           assigner, λX: can(X, task)))
```

that is, the assigner knows that the informer knows some actor that can perform the task, so the assigner asks the performer to tell it.

Alternately, the assigner may ask the informer to perform the task assignment, as discussed in section 5.6.

5.2.3 Market-like assignment

Finally, the assigner may believe that some actors are capable of performing the task, but not know which actors in particular. In this case, the assigner can ask many actors, only some of whom are possible performers.

There are two possible requests. First, the assigner can ask the actors if they can in fact do the task; the assigner then chooses one of the actors and assigns the task. This model is used in a market-like setting, where the assigner sends requests for bids to potential performers and those who are able to perform the task respond with bids. It also describes the situation where an assigner "asks for volunteers."

In Car Co. and Airplanes, Inc., the task of manufacturing a subcomponent is frequently put up for bids; manufacturers interested in performing the task submit bids.

I represent the recipe in this situation as follows:

```
want(assigner, task)

know(assigner, ∃X ∈ actors: can(X,task))
```

The application of this recipe leads to the following communication:

```
∀X ∈ actors: request(assigner, X,
    informtrue(X, assigner, λX can(X, task))
```

that is, the assigner wants the task performed and knows that some actor in the set `actors` can do it. The assigner therefore asks each of the actors to say whether or not they can do the task. When the actors reply, the assigner will `know(assigner, can(performer, task))` for some set of performers.

Second, if the assigner does not care how many performers actually do the task, it can simply broadcast the description of the task to all possible performs. Each performer evaluates the task and performs those it chooses to. (Note that some tasks may be performed repeatedly and others not at all.)

This situation can be represented as the following recipe:

```
want(assigner, task)

know(assigner, ∃X ∈ Actors: can(X,task))
```

and communications action:

```
∀X ∈ Actors: request(assigner, X, task)
```

The major advantage of the final two mechanisms discussed is that the pool of possible performers is increased, potentially reducing the cost of performing the task.

However, the coordination mechanism has a higher cost. First, sending messages to multiple actors may be expensive, even if there is some way to broadcast a description of the task to numerous actors. Second, each potential performer must expend some effort to evaluate if they can perform the task and prepare a bid.

5.3 Choosing an actor to perform the task

If there are multiple performers identified in the previous step, the assigner must choose one in particular to do the task.

In the simplest case, there is only one potential performer, that is, only one actor that can perform the task. Often this may be due to specialization of roles of departments or individuals (i.e., who is supposed to do a particular tasks) than to a distribution of ability (i.e., who is capable of doing the task).

In all sites, all drafting was done by the drafting room, thus eliminating the need to decide who should prepare the final drawings.

In Car Co., some components were manufactured by a subsidiary company. In these cases, the parts would generally not be put out for bid; instead, the subsidiary would be given the contract.

In the general case there may be several actors who could do the task making it necessary to choose one. To choose a particular actor or other resource from those available requires some knowledge for evaluating how good a particular actor will be at performing the task. There are many possible bases for making this decision; here I will only try to suggest a few of the more common. Which one to use depends on the nature of tasks being coordinated. The knowledge necessary can be represented abstractly, as the capability to take a list of potential performers and somehow choose one.

```
pick-best-actor(Performer, Actors)
Adds:          know(Performer, best(Actors))
```

One basis for such a decision is to choose a technique that reduces the cost of making the decision, such as picking the first actor found or choosing randomly from the actors presented. If the task is performed repeatedly, a possible strategy is to always pick the actor that performed the task previously. This strategy has three advantages: first, as mentioned before, it eliminates the need to search for alternative performers; second, it eliminates the need to choose

between already known performers; and third, the performer may become more experienced at performing the task.

In Car Co. and Airplanes, Inc., when a part is redesigned, rather than attempting to find a new supplier to manufacture the new part the contract of the current supplier is often extended.

Otherwise, the assigner must have some evaluation criteria to determine which actor will perform the task "best" or at the lowest cost. Using these heuristics requires additional knowledge, such as to what other tasks a particular resource has been assigned or the relative values of different resources.

In Computer Systems Co., a call processing system is used to assign calls to the next free call handler in the response center to keep the load on the actors balanced and thus reduce the overall response time.

In Car Co. and Airplanes, Inc., some parts are put up for bids. The bid returned includes the cost of providing the part, and the lowest bid can be selected.

5.4 Getting a chosen actor to perform a task

Once the best actor to perform the task has been identified, the assigner must get that actor to actually perform the task.

For the present study, I assumed that actors accept their roles within the organization and will therefore accept legitimate requests, making it unnecessary to convince the actor that it should perform the task. If this assumption were relaxed, then this step would have to include considerations of the possibly different interests of the actor.

In some cases, the notification may happen as a side-effect of an earlier step. For example, if the performer chooses itself to do the task, as discussed below, then presumably it already knows to do the task.

Otherwise, this assignment requires some mechanism for communicating with the performer. In the sites studied there are two basic mechanisms for this

notification. First, the assigner may send some kind of task assignment message to the performer.

In Car Co., the purchasing department sends contracts to the chosen suppliers.

In all sites, engineers receive problem report messages or change requests and send work orders to designers or drafters.

This situation can be modelled as the following recipe:

```
want(assigner, task)

know(assigner, can(performer, task))
```

that is, the assigner wants the task do and knows that the performer can do it. Therefore, the assigner simply requests that the performer do the task:

```
request(assigner, performer, task)
```

By the definition of the request action (in Chapter 2), this results in the performer having a goal of performing the task.

Second, the assigner may make some kind of change to shared objects which is interpreted by the performer as a signal to do some task.

In Car Co., when an engineer changes a part, the new part description is entered in a parts database. These changes are picked up by the purchasing department as an indication that they should find a supplier to start making the new parts and by materials management, that they should plan for the change.

In terms of the representation, this mechanism is similar; the assigner performs some action which results in the performer having a goal of performing the task. The exact axiomatization will depend on the details of the shared object.

In order for the performer to perform the task, the assigner must have some way to describe the task. I have assumed that the actors share a common language in which the task can be described, but developing such a language

may be problematic, especially when tasks are assigned across organizational boundaries.

5.4.1 *Disposition of results*

The description of the task includes what to do with the result. The original actor may or may not care about what happens to the result; it might ask that the result be returned to it or be sent to someone else.

We can represent the former as an additional request:

```
request(assigner, performer, informref(performer,
          assigner, λX: X=result(task))
```

that is, the assigner asks the performer to tell it the result of the task.

5.5 Assigning resources

Having chosen a resource, an action may need to ensure that no other action will use it at the same time. The simplest method is for each actor to that wants to use the object to to check if it is being used and only try to use it if it is unused.

The status of the object can be checked either directly or through some kind of locking mechanism, like the fact that the code for the module is signed out of the code library. Either case can be modelled as an action:

```
check-if-busy(Performer, Object)
Adds:          know(Performer,
               ∃Action: use(Object, Action, now) v
               ~∃Action: use(Object, Action, now))
```

To actually do the assignment means changing the status of the object so other actors see that it is busy.

This assignment may require some kind of mechanism for choosing which tasks gets the resource if multiple tasks are waiting for it. For example, a queue

waiting for a resource ensure that only the head of the queue will try to use it next; assigning numbers to people waiting for service and using a sign for the next in line serves the same function.

Alternately, the tasks may be assigned a particular time during which they can use the resource. Such a mechanism is much simpler if the assignment is centralized in one actor. An actor that wants to use an object sends a request for it to this actor and waits for a reply saying that it has been assigned.

5.6 Assigning coordination tasks

Performing a coordination method is a task that may itself be assigned to some other actor to be performed. This assignment may occur using the methods described here and may occur at any stage in the process.

Typically assigning the task assignment task is easier than simply performing it because of the specialization of actors' roles. Commonly, the entire process of identifying, choosing and notifying the performer is assigned.

In Computer Systems Co. and Airplanes, Inc., customers do not need to know how to assign a problem report to a particular engineer because they can simply send all problem reports to another actor (in Computer Systems Co., the response center; in Airplanes, Inc., a customer engineer).

This situation can be modelled as the following recipe:

```
want(assigner, task)

know(assigner, can(performer, task))
```

The application of this recipe leads to the following communication:

```
request(assigner, performer, task)
```

that is, the assigner wants the task performed and knows that the performer can do it, so it asks the performer to perform the task. Note that the assigner's model of the performer is quite independent of the performer's actual capabilities. As

in the example, the performer may perform the task by assigning it to someone else.

Alternately, the assigner may know that the task will be reassigned. For example, in the case where an informant knows which actors can perform some task, the assigner may send the task to the informant who forwards it to the performer himself.

In Airplanes, Inc. the engineering change control group receives tasks and forwards them to the next person to do something.

This situation can be modelled as the following:

```
want(assigner, task)

know(assigner, ∃Actor know(informer, can(Actor,
   task)))
```

The application of this recipe leads to the following communication:

```
request(assigner, informer, task)
```

that is, the assigner wants the task performed and knows that the informer knows some actor that can do it. Therefore, the assigner asks the informer to perform the task, knowing that the informer can ask someone else.

Sometimes the actor first attempts to perform the task and assigns it if it can not do it.

In Computer Systems Co., if the marketing engineer can not locate a reported bug, then the problem is sent to the bug tracking team for further investigation.

In Airplanes, Inc., the liaison engineers can fix minor problems themselves but refer problems that they can not quickly diagnose to the design engineers.

This can be modelled by assuming that the actors have an action that can perform some of the tasks and which has a lower cost than asking someone else.

In this case, the actor will first try to perform the action; if it fail, they will then try the second action, that is, asking someone else.

Sometimes only part of the task assignment process is assigned. One can imagine a situation where one actor determines which actors could perform the task, presents the list to a second actor who picks one of them and gives the choice to a third actor who notifies the selected actor.

In some cases, performers may perform some of the task assignment process themselves. For example, there might be a queue or set of tasks waiting to be performed; free actors simply take the first task on the queue or look through the set until they find a task they can perform.

In Computer Systems Co., customer calls are placed a queue; the response centre specialists can choose which calls to answer, although their overall performance is monitored.

6 Category 3: Dependencies between objects

The third category in the typology is dependencies between different objects. In this domain, an important set of dependencies arises from parts that physically touch each other or fit together. If two tasks use different objects that are interdependent, then the two tasks are interdependent as if they both depended on a common object.

Depending on the level of decomposition of the objects, two tasks may appear to be using a common object because they are each using different parts of a more complex object. For example, two tasks may both appear to consume an orange, because one requires the rind and the other the pulp.

To manage these interdependencies, actors must first identify that they exist and, second, decide what to do about them. For the task I studied, "what to do" is to possibly change the interdependent parts to match changes to other parts. The final example discussed in the introduction, engineers discussing changes with the engineers responsible for other modules, who possibly make additional changes to those modules, falls in this category.

6.1 Identifying interdependencies

The procedures discussed in section 4 for determining when two tasks use the same object must be extended to manage interdependent objects. For example, an important part of developing a change to a part is identifying interactions between the part being changed and other parts.

One technique for locating interdependencies are the engineers' mental models of the product and the organization.

> *In all three sites, engineers usually know with which parts their parts interact and with which other engineers they therefore need to discuss a change, either through past experiences or training.*

Problems often occur when a dependency is for some reason overlooked, such as the fatal system errors caused by the word processor at Computer Systems Co.

These mental models can be represented by the engineer knowing about a set of relationships between the components. For example, the fact that one module uses an interface provided by another might be represented by:

$$\texttt{know(actor, uses(module}_1\texttt{, module}_2\texttt{))}$$

In addition, all three sites use various sorts of physical models of the product as a mechanism by which engineers can determine the effects of a proposed change.

> *In Computer Systems Co., engineers use test systems (computers running the latest release of the operating system) on which a changed module can be added and tested.*
>
> *Car Co. and Airplanes, Inc. use various forms of mockups to allocate space for new parts or to check for interactions with existing parts.*

These physical models can be consulted to determine if there is an interdependency. We can model this consultation with an action, such as:

```
check-model-for-interdependencies(Performer, Model
   Component)
Adds:          ∀Other-component ∈
               components(Model):
               know(Performer, physically-
               interdependent(Other-component,
               Component))
```

In all three cases there is a problem keeping the models up-to-date. A new part may appear to work or not work because the model has earlier versions of some other parts.

6.2 Evaluating effect of interdependencies

Once the dependency has been identified, the effect of the change on these parts must be evaluated. Identification of a dependency by some actor results in that actor knowing about a new task of checking for problems. These tasks could be then assigned to some actor, using the processes discussed above.

In all three sites, an engineer planning to make a change consults with the engineers responsible for the other parts. Those engineers must perform additional tasks to ensure that their parts will not be affected by the proposed change or to determine the need for another change.

We can model this process of task creation by a rule such as:

```
know(actor, plausible(want(actor,
   change(component_1)) ∧ physically-
   interdependent(component_1, component_2),
   goal(actor, change(component_2))))
```

Note that this is actually only a plausible inference; many changes may not affect other components.

Second, additional work may be necessary to ensure that desired relationships are maintained. For example, there may be many physical copies of

a single logical object, such as a manual; there are many copies of the manual, but the information in those manuals should be the same. However, if the information is updated, the physical and logical objects may get out of step, requiring additional work to bring them into agreement. This problem can be especially difficult if there are simultaneous updates can be made to the physical copies, in effect creating multiple versions of the object.

7 Related work

Given the importance my definition of coordination places on dependencies, I will briefly review some related work in organizational behaviour and artificial intelligence on different kinds of interdependencies.

7.1 Interdependence in organizational behaviour

Dependence has been considered an important reason for needing coordination and many researchers have investigated ways of characterizing it. In organization theory, dependencies are usually considered to arise between actors, either individuals or work units.

7.1.1 Dependency as the inverse of power

Dependency has been viewed as the inverse of power, where power is the ability of one actor to control the outcomes of another. For example, in Emerson's (1962) view, an actor A depends on an actor B if A "aspires to goals or gratifications whose achievement is facilitated by appropriate actions on B's part" (p. 32), but he did not discuss how this concept could be measured. He did suggest four ways to reduce the tensions of power relationships, such as withdrawal of actor A from the relationship or development of alternatives to actor B, but did not consider situations in which the imbalance is maintained.

Victor and Blackburn (1987) characterize the actions available to each actor by considering a pay-off matrix showing the outcomes for each actor for each combination of decisions by the two actors. They note three possibilities: the outcome for an actor could depend entirely on the action taken by the other

actor, jointly on the actions taken by both units, or only on its own actions. They therefore define dependence as the ratio of the sum of squares of outcomes controlled by or dependent on another actor to the sum of squares of all outcomes. They also define the degree of conflict between the actors by looking at differences in desired outcomes.

These two conceptualizations of dependency provide a way to measure the strength of the relationship between two actors but they do not suggest what must be done to manage it. They seem to be focused exclusively on the one-time effects of actors' decisions rather than on-going processing of tasks. Furthermore, it is not obvious that it is feasible to obtain empirically the various outcome values or even the possible decisions.

7.1.2 Topologies of interdependence

Thompson (1967) categorizes dependency in terms of work flow between actors. He describes three kinds of dependencies: *pooled*, where units get inputs and contribute outputs to the organization independently; *sequential*, where the output of one unit is input to another; and *reciprocal* where one unit's output is also its input, possibly through a series of other units. Thompson suggests that organizations should use different coordination methods for each kind of dependency: for pooled, organizations coordinate by standardization; for sequential dependency, by plans; and for reciprocal dependency, by mutual adjustment.

Van de Ven, Delbecq and Koenig, (1976) attempt to empirically test some of Thompson's ideas. They suggest a fourth dependency to add to Thompson's three: *team dependency*, where actors work together to complete a task with no "measurable temporal lapse in the flow of work" (p. 325). They examine three categories of coordination mechanisms to manage these dependencies: impersonal, which includes various kinds of programming, such as rules, plans or schedules; personal, such as horizontal or vertical individual contacts; and group, such as planned or unplanned group meetings.

In a questionnaire study of 197 units of a large state employment security agency, they find that as the degree of dependency between members of the unit increased, so did the overall use of coordination mechanisms; in addition, the use of group and personal coordination mechanisms increases, while the use of impersonal coordination mechanisms decreases.

One problem with Thompson's characterization of dependency is that in some ways the decision about how to arrange the work flows is itself a choice of a coordination mechanism as opposed to a necessary characteristic of the task. For example, given a task to be performed repeatedly by a group of actors (e.g., processing change notices), one can either break the task into small subtasks (e.g., checking the part numbers are correct, checking the availability of new parts, etc.) and have each actor perform one subtask for all tasks (as on an assembly line) or assign complete tasks to different actor to perform. Thompson's work is thus more accurately described as a typology of topologies of work flows.

7.1.3 Dependence depends on exchange of resources

McCann and Ferry (1979) take a view of dependence similar to Thompson. However, they focus on the exchange of resources between actors. They characterize these exchanges along six dimensions: 1) the number of resources exchanged; 2) the amount of the resources exchanged per unit time; 3) the frequency of exchanges; 4) the amount of time before loss of the resource has an effect; 5) the value of the resource, including the cost of substitutes, the cost of locating a substitute, the importance of the resource and the percentage of time actor A's needs were satisfied by actor B in the past; and 6) the direction of resource flow, to, from or both ways between actors. They suggest that dependency increases as these dimensions increase. They do not, however, suggest specific ways to manage dependencies, aside from noting that managers of interdependent work units should ensure that both units have same perceptions of the level of dependence and that the perception is accurate.

7.2 Dependence in AI research

Researchers in distributed artificial intelligence consider what kinds of dependences arise between plans or goals.

7.2.1 Goal relationships

Wilensky (1983) suggests a taxonomy of goal relationships (p. 51) and distinguishes between negative and positive relationships. Negative goal relationships include resource limitations, mutually exclusive states and what he calls preservation states (e.g., wanting to take a day off vs. wanting to keep one's job). Resources include time (deadlines or need to synchronize with an external event), consumable functional objects, nonconsumable functional object and abilities. Positive relationships include mutual inclusion (the planner has the same goal for more than one reason) and plan overlap (where a single action satisfies several goals).

Ways of resolving conflicts include replanning (i.e., looking for a way around a problem), trying to change the circumstances (e.g., getting rid of the deadline, changing the timing of the external event, increasing the capacity of a resource) and fully or partially abandoning a goal.

7.2.2 Plan relationships

Several researchers consider possible relations between goals and actions when trying to merge plans. In this model, the two plans are independently generated and then merged to create a single plan for both actors that avoids conflicts and takes advantage of possible synergies. In other cases, the input may be a single plan that includes multiple actors. For example, Stuart (1985) describes a system that took a multi-actor plan as input and inserts synchronizing operations to eliminate conflicts between different operations. If one operation B has a particular precondition that is set by another operation A, the resulting plan will require that operation A be executed before operation B.

von Martial (1989) suggests both negative and positive relationships between plans executed to achieve different goals. Negative relationships

include what might be called outcome conflicts, where the goals of the two plans are logical inconsistent and can not be simultaneously satisfied, and resource conflicts, where the plans together require more of a resource than is available. He further distinguishes between consumable and non-consumable resources, noting that conflicts over non-consumable resources—such as two plans requiring the same tool—can be resolved by adding some kind of locking around or scheduling the use of the resource.

Positive relationships include equality, where two plans happen to require the same actions, and consequence relationships, where a desired subgoal of one plan is the result of the other plan; thus both plans can be satisfied by having only one actor do those actions. von Martial (1989) also discusses what he calls favour relationships, where a slight modification of one plan accomplishes a goal of the other plan. The modified plan requires additional work on the part of one actor, but much less work in total. (For example, if both actors plan to go to the post office, one to mail a parcel and the other to buy stamps, both plans can be achieved at lower total cost by having one actor do both errands.) In addition, von Martial considers requests as a kind of dependency, since satisfying these requests requires modifications to the other actor's planned actions.

Decker and Lesser (1989) are also interested in coordinating actors by merging partially shared plans. They characterize the possible relationships between goals in a goal hierarchy, in which a goal is expressed as a conjunction or disjunction of subgoals. They point out that if there are no interactions between the subproblems, then there is no need to cooperate.

Based on a consideration of the goal hierarchy, they suggest four generic categories of relationships: graph relations, temporal relations, domain-dependent relations and resource relations. Graph relations are derived from the graphic representation of goals and subgoals; these might be called syntactic relationships. For example, one goal might be a subgoal of another; if two goals have equivalent subgoals then they are equivalent. Temporal relations are derived from the start and finish times or durations of actions and include before, after, overlaps, etc. Domain-dependent relations include relations such as inhibits, cancels, enables, constrains, etc. These might also be called semantic

relationships. For example, goal A inhibits goal B if when A is achieved, B can not be achieved. Domain-dependent relations are domain dependent because they can be evaluated only with respect to particular domain but are generic because they arise in many domains. Finally, goals may be linked by resource constraints; their system does not address these relations but they recognize that they exist.

7.3 Conclusion

Most authors seem to agree that coordination is necessary to solve problems caused by interdependences. Most researchers in organizational behaviour consider interdependencies between work units. The notion of interdependence, however, is not well defined or obviously operationalizable. Thompson (1967), for example, suggests three different kinds of interdependence but he does not really indicate how you should identify these in a real organization, apart from the nature of the interactions. This lack is most apparent in his examples, since it is unclear from his discussion which came first, the organizational structure or the interdependences. In particular, one might wonder whether Thompson could have identified suboptimal organizations where the interdependences existed but were not reflected in the organizational structure.

Because I consider the choice of which actor should perform a action part of the task of coordinating, I believe dependencies are best thought of as between actions rather than between work units. Once the actions have been assigned to particular actors, then the dependencies may appear to be between those actors.

I think the AI researchers are on the right track in considering possible relationships between goals and actions rather than work groups and many of the interdependencies they identified can be found in my typology. However, the lists of possible relations generated in early research seem some what *ad hoc.* I have attempted to address this issue by basing my typology on possible relationships between the tasks and objects and on the use of common objects.

8

CONCLUSION

"Tut, tut, child," said the Duchess. "Everything's got a moral if only you can find it."

—Lewis Carroll, *Alice in Wonderland*

In this chapter, I summarize my findings and show how the modelling technique and typology developed in this thesis can be used to solve the benchmark problems posed in the introduction. I conclude by discussing some ideas for future research.

1 Summary of results

In Chapter 1, I begin my study of coordination by describing some of its properties. I then offer a better definition of coordination in terms of coordination problems caused by interdependencies between elements of a coordinated situation—actors, actions, goals and objects—and coordination methods performed to address those problems.

In Chapter 2, I present a technique for modelling coordination methods, based on approaches to modelling autonomous actors developed by researchers in distributed artificial intelligence. A recipe is defined as what someone knows when the they know how to do something. I represent what they know in an augmented first-order logic. In particular, I model individuals' goals, capabilities and knowledge and show how an actor attempting to achieve its goals communicates with other actors.

I present my methodology for studying and modelling organizations in Chapter 3 and describe how I use these techniques to model real organizations

performing a coordination-intensive task, namely engineering change management. The results of field work in 3 organizations is presented in Chapters 4, 5 and 6, along with models of parts of the engineering-change management process involving the design engineer.

In Chapter 7 I develop a typology of coordination problems and use it to organize the coordination methods found in the case sites. The typology is based on kinds of interdependencies that arise between elements of a coordinated situation. I group the initial four elements—actors, actions, goals and objects—into two, tasks and objects. Tasks include both performing an action and achieving a goal; objects include all things affected by actions, including, as a special case, the effort of actors. This results in a three element typology, interdependencies between 1) tasks and tasks, 2) tasks and objects and 3) objects and objects.

Interdependencies between tasks and tasks are divided into task-subtask relationships and interdependencies between otherwise unrelated tasks. Problems in the first category can be handled, for example, by task decomposition.

I further divide the second category, interdependencies between otherwise unrelated tasks, by looking at the ways two tasks can be interdependent. I conceptualize this interdependence in terms of common objects, that is, some object that is created or used in some way by both tasks. This conceptualization results in another three part division: task-task interdependencies can be create-create, create-use or use-use.

For example, if two tasks both create a common object, then it may be possible to merge the two tasks or perform only one of them, possibly saving effort. (This category includes the first of the examples of coordination presented on pages 20–21, checking if a reported problem duplicates a known one to avoid duplicate problem reports.)

Task-object interdependencies, the second category of my typology, include assigning a resource or an actor to a task. (This category includes the

second of the examples presented on pages 20–21, the problem of routing a problem report to a software engineer who can fix it.) In the thesis, I primarily considered problem of assigning an actor to a task, but process of assigning a resource is parallel. I present a four-step framework for task assignment, 1) identifying task requirements, 2) identifying actors who could work on the task, 3) choosing the best actor and 4) getting that actor to perform the task.

For example, difference between hierarchical and market assignment mechanisms is way that the assigner identifies possible actors. In hierarchical assignment, the assigner already knows which actors are available; in market assignment, the assigner finds out by asking them.

The final category of the typology is object-object interdependencies. The problem here is of finding and managing interdependencies between objects. In the case of engineering changes, these interdependencies arise because a change to one part of a product may result in the need for a change to other interdependent parts. This category includes the third of the examples presented on pages 20–21, the need for software engineers to consult other engineers when making a change to a module.

2 Solutions to benchmark problems

In this section, I will discuss how the modelling technique and typology of coordination problems developed in this thesis can be used to answer the benchmark questions I posed in the introduction to this thesis. I will first briefly recap the problems and indicate how I solve them.

First, in what ways can a given organization be arranged differently while achieving the same goals? As a concrete example, I discuss the way problem reports are assigned to software engineers in one of my case sites and consider alternative ways the engineers could have been organized.

Second, what kinds of information technologies might be useful to support a particular organization? I use the modelling technique to suggest a system to support software engineers working on problem reports.

Finally, how might the preferred structure for an organization change with extensive use of information technology? I briefly show how information systems differentially affect different organizational forms, possibly changing which form is preferred.

2.1 Organizational redesign

A strength of the theoretical framework developed in this thesis is that it allows me to go beyond the three sites I studied to a more general coordination level. Understanding the coordination problems addressed by the organization suggests alternative coordination methods that could be used.

For example, many of the actors in the Computer Systems Co. case perform some part of a task assignment process. The response centre distinguishes problems by product and assigns them to an appropriate marketing engineer; the marketing engineer assigns problems to a particular group; and the group contact, to a particular engineer. Such a process is necessary because engineers are specialized by module—only a single actor can fix a particular problem—and the task assignment problem is to find that particular actor and get it to fix the problem.

In other parts of the company, including, apparently, the support group for released versions of the operating system, programmers are generalists and multiple actors can fix a problem. I did not study this group in detail, but it seems that tasks are assigned to actors based on load; the first free actor takes the next task on the list.

This alternative organization suggests the possibility of some kind of market-like task assignment system for problem reports. In this model, a description of each problem report would be sent to all available engineers. Each engineer who could fix the bug would prepare a bid, saying how long it would take to fix the bug, how much it would cost or even what they would charge to do it. The lowest bidder would be chosen and the task assigned to him or her.

The analysis in chapter 7 shows what the actors need to know to perform this process. First, assigner needs to know which actors are potential performers and needs to be able to communicate with them. Second, there must be a common language in which the tasks can be described. Finally, the assigner need some way to evaluate the bids returned.

The performers need to be able to evaluate each task and estimate how long it would take to fix the problem (or how much it would cost) and to communicate with the task assigner.

2.1.1 *Evaluating alternative organizations*

Given this framework, we can evaluate the advantages and disadvantages of each kind of organization. I have not fully worked out ways of evaluating these costs and benefits, so in this section, I will simply suggest informally how such an evaluation might proceed.

To compare the organizations, we must first decide what it is the organizations do, in order to evaluate how well they do it. The model I will adopt here is that of a task processor. Tasks arrive at the organization, needing some kind of processing. The organizational problem is to assign the tasks to some actor that can execute them. (This is by no means the only or even most appropriate model, but it does have the advantage of simplicity.)

Given this model, there are many possible objective functions, including, for example, the number of messages sent in each model (Malone, 1987), the total processing time for each task, the total cost of different organizations with the same capacity, the coordination cost (i.e., the cost of the task assignment process) and the production cost (i.e., the cost of performing the task) in each organization, or the organization's vulnerability to the failure of an actor (Malone, 1988). Many other factors could be added to complicate such a model, such as organizational learning or individual motivation. At present, this list of factors seems quite ad hoc; further research may provide a better framework for evaluating alternative organizations.

Obviously, any of these comparisons depends on many assumptions about, for example, the relative speed of actors, the number of actors, the communications paths, etc. In principle, most of these values could be derived from the case study. Rather than make these assumptions explicitly, however, I will simply compare the organizations qualitatively.

Viewed this way, the current organization has a lower cost of coordination; assigning a task involves sending only three messages. However, it is vulnerable to the failure or overloading of a single actor (the engineer responsible for each module has no backup). Because each actor is a specialist, presumably he or she will be able to fix problems relatively quickly. However, if the load is distributed unevenly (i.e., some modules have more problems than others) then a problem may have to wait until the appropriate specialist is free, thus increasing the total time to finish a task. Also, some specialist may be underutilized, increasing the cost of the whole organization without increasing its performance.

The cost of the task assignment in the generalist model is also low. Furthermore, problems are handled by the next available actor, minimizing waiting time and reducing vulnerability to failure. However, because each actor is a generalist, the time he or she takes to fix a module is likely to be higher than in the specialist model. Furthermore, since the tasks any actor performs are randomly distributed, the organization takes no advantage of any difference between actors in performance and no actor has much opportunity (or incentive) to learn to improve.

The market-like model has a much higher coordination cost, requiring many messages for each task (one for each bid request and bid). (The cost of processing these messages includes, for example, the cost of having each engineer read each problem report.) However, problems can be immediately assigned, again reducing the waiting time. Furthermore, in this model, the task will be assigned to the actor with the lowest bid, thus taking advantage of differences in ability. If the actors learn, then can specialize, preferentially bidding for one type of task and constantly improving their performance on it.

2.1.2 *Apprentice system for organizational design*

If better formalized, the kind of analysis sketched above could be incorporated into an expert support system. Such a system, given a task to be performed and the set of actors available, would suggest alternative ways to organize the actors or work out the implications of particular organizational designs. Based on the recipes for different coordination methods, the system would suggest what coordination work each agent would do and even what kinds of support systems would be useful. A possible output of such a system is a set of Object Lens (Lai, et al., 1988) templates and message processing rules for each actor, designed to support the kinds of coordination methods that role actor.

2.2 Designing computer-support systems

One possible use for a model of an organization is to suggest possible information systems to support the members of a group performing the task by suggesting what information should be provided to help them coordinate.

Such a tool may be especially valuable for new uses of information systems. Most current uses of information systems have been designed to completely automate some function (e.g., a payroll or accounting system) or to support a single individual working alone (e.g., a word processor or a spreadsheet). For technologies such as electronic mail or inter-organizational order processing systems, however, such a view is unhelpful. It may be more useful to think of such systems as supporting some coordination process.

For example, one coordination problem faced by engineers making changes is determining which other engineers might be affected by a proposed change. An information system could help by notifying them of interactions with other parts they might otherwise overlook or by helping them find the engineer responsible for a particular part.

Currently, engineers seem to independently and informally keep track of which other engineers' parts interact with the parts they maintain. This kind of information is easy to maintain manually and in a decentralized fashion when

the development group is small and physically close together but the task becomes more difficult as the group grows and as other, more remote groups are involved.

One way to implement such a system would be a database of interfaces between parts and users; an engineer planning to modify an interface could use the database to determine who should be notified. However, such a database might quickly become out of date. If so, it would be no worse than the current system but would offer few advantages and would therefore probably not be used. (In fact, one of our interviewees in the Computer Software Co. had developed a database of interfaces, but had decided not to use it until a better system could be devised to keep it up to date.) A more useful system would include better methods and motivations for users to register their use of interfaces. For example, having a tool that noticed when a new interface was used might make it easier for an engineer to know to register as a user of the interface. For computer software, a system could compute the interdependencies directly from the code of the operating system, thus guaranteeing a complete and accurate list.

2.3 Effects of intensive use of information technology

The use of information technology may differentially affect the costs of different organizations. For example, in the market model, the response center needs to send the same message (bid request) to multiple actors. To support this communication, the organization could use some kind of computer bulletin board on which task announcements could be posted. Such a system could cut the coordination cost in half by replacing all bid request messages with a single broadcast. Processing of bids could also be automated, further reducing the cost. By contrast in the other organizations, such a system would only speed the transmission of messages, not reduce the number required.

3 Future work

In this section I will briefly describe some further research suggested by the modelling technique and the typology.

3.1 Where do goals come from

First, I am interested in developing methods for identifying an organization's goals. For the purposes of this study I specified the the goals in which I was interested and studied how the organization achieved those goals. As a result I did not consider alternative goals the organization could have chosen to pursue or how the particular goals were chosen. In a more general model, we would also include how goals are chosen by the organization.

I also did not consider the interaction of organizational and individual goals. An extended model would include incentives and a model of convincing actors to do something.

3.2 Development of coordination knowledge

Second, I would like to study how coordination methods develop. In the current study, I assumed actors knew how to coordinate and focused on what knowledge was necessary and what made it necessary. A possible extension, therefore, is to study the origins of these methods.

There are many possible sources for these methods, such as training, previous experiences, observing or asking coworkers or negotiation with other groups. One possible strategy for studying changes in coordination methods is to examine how organizations react to generational changes (Henderson and Clark, 1989).

3.3 Computer simulations

Third, I hope to use the modelling technique as a basis for developing computer simulations of organizations. Such simulations would provide a powerful tool for exploring the implications of different organizational structures and distributions of knowledge and capabilities.

Two key issues that need to be solved in implementing these models are the representation of the actors' knowledge and efficient ways of implementing their reasoning.

3.4 Presentation of the models

Finally, I am interested in developing more concise representations of the models. As currently presented, they are not very easy to read. Furthermore, the methods used to generate alternative organizations are some what ad hoc. One approach to simplifying these models might be to use inheritance and specialization so that only the differences between similar processes would have to be noted. One possibility is to use the work on abstraction in planning (Tenenberg, 1988) to represent coordination at different levels of abstraction. Organizations might therefore be similar at some abstract level and yet different in particulars.

For example, consider the coordination necessary for a group of actors (see Figure 8.1) to cooperatively perform some task. This is an example of a

Figure 8.1. A collection of general purpose (circles) and specialized actors (triangles and squares).

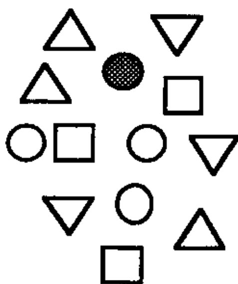
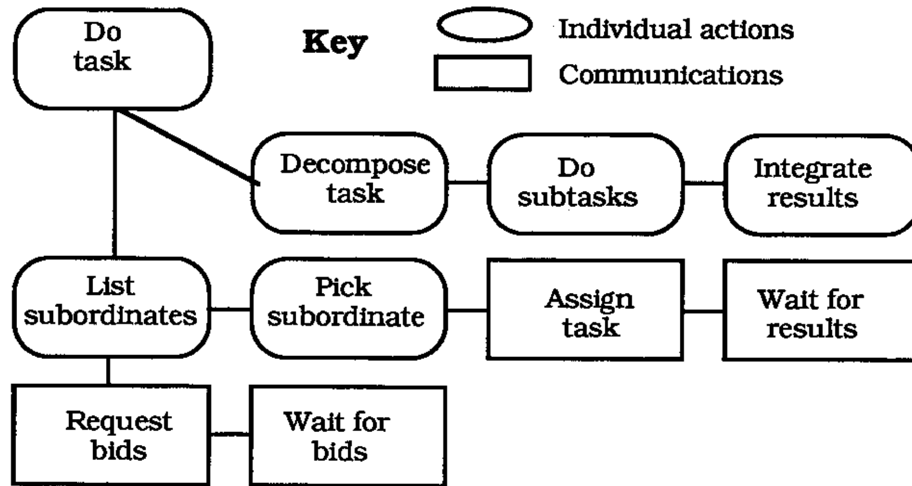


Figure 8.2. Alternative ways of performing a task: with or without task decomposition and with no, hierarchical or market task assignment.

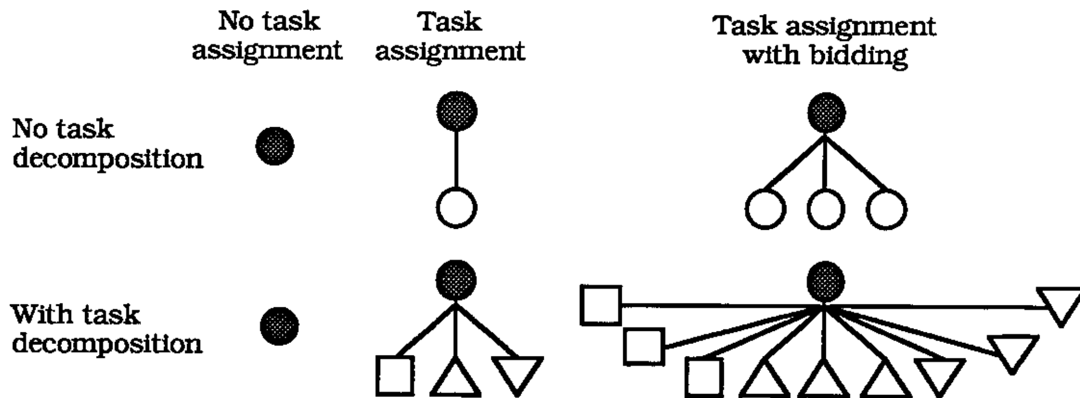


particular generic coordination process, namely the process of task assignment. One actor (the dark circle in the figure) has a task to be done.

The actor can perform the task in several ways. First, it can simply do the task itself. Second, it can decompose the task into subtasks, do each of the subtasks and integrate the results. Finally, it may assign the task (or one of the subtasks) to one of the other actors, either the entire task to a general purpose actor (the other circles) or specific subtasks to specialist actors (the triangles and squares). To determine which other actors are available to perform a task, the actor can either keep track of the status of all actors and picking an appropriate one (as in a hierarchical organization) or request actors to bid on the task (as in a market). These options are summarized in Figure 8.2.

While simple, this model gives rises to a number of distinctive communications patterns. For example, if an actor always follows the same process when assigning tasks and subtasks (e.g., always uses bidding or never decomposes tasks) there are six different organizational forms that will result, shown in Figure 8.3. Since each subordinate actor can follow this process, complex organizational forms may quickly emerge. At the highest level of

Figure 8.3. Organizational structures resulting from different combinations of the abstract steps above.



abstraction, these organizations are the same, since they have a common goal; they differ only in the details of how they achieve that goal.

A reanalysis of coordination mechanisms suggested by earlier researchers in terms of the information processing view may suggest common patterns of communication that could be used to simplify the models. A second way to simplify these diagrams is to use inheritance and specialization so that only the differences between similar processes would have to be noted.

4 Conclusion

I began this thesis by asking, "What good are organizations?" My answer was, essentially, that organizations are ways of coordinating the actions of multiple actors to achieve common goals. As new technologise make possible new ways of coordinating, understanding what individuals in organizations do to coordinate becomes increasingly important.

This thesis has presented the first steps towards a theory of coordination. I have developed a technique for modelling coordination methods, based on ideas from distributed artificial intelligence, and a typology of coordination methods found in three case sites.

Much remains to be done, but even the initial results of my thesis should be useful in several ways. A better understanding of how individuals work together may provide a more principled approach for designing new computer applications, for analyzing the way organizations are currently coordinated and for explaining perceived problems with existing approaches to coordination. By systematically exploring the space of possible coordination strategies, we may be able to discover new kinds of organizations—organizations in which humans and computers work together in as yet unimagined ways.

References

It is not all books that are as dull as their readers. ... These same questions that disturb and puzzle and confound us have in their turn occurred to all the wise men; not one has been omitted; and each has answered them, according to his ability, by his words and his life.

—Thoreau, *Reading, Walden*

Aerospace takes off. (1989, April 8). *Economist*, pp. 71–72.

American Heritage Dictionary. (1981). Boston: Houghton Mifflin.

Ancona, D. G. and Caldwell, D. F. (1990). *Demography and design: Predictors of new product team performance* (Unpublished paper). Cambridge, MA: MIT Sloan School.

Austin, J. L. (1962). *How to Do Things with Words.* Cambridge: Harvard University.

Barker, B. (1989). *The commercial aircraft industry: Boeing.* Presentation for "America's competitive challenge in the 90s", October 2, Cambridge, MA: MIT Sloan School.

Barr, A. and Feigenbaum, E. A. (Eds.). (1981). *The Handbook of Artificial Intelligence* (Vol. I). Los Altos, CA: William Kaufmann.

Beer, S. (1967). *Cybernetics and Management* (2nd ed.). London: English Universities Press.

Bernard, H. R., Killworth, P. D. and Sailer, L. (1980). Informant accuracy in social network data, IV: A comparison of clique-level structure in behavioral and cognitive data. *Social Networks*, 2, 191–218.

Bernard, H. R., Killworth, P. D., Kronenfeld, D. and Sailer, L. (1985). On the validity of retrospective data: The problem of informant accuracy. In *Annual Reviews in Anthropology*. Palo Alto: Stanford University Press.

Bond, A. H. and Gasser, L. (1988a). An analysis of problems and research in DAI. In A. H. Bond and L. Gasser (Ed.), *Readings in Distributed Artificial Intelligence* (pp. 3–35). San Mateo, CA: Morgan Kaufman.

Bond, A. H. and Gasser, L. (Eds.). (1988b). *Readings in Distributed Artificial Intelligence.* San Mateo, CA: Morgan Kaufman.

Boulding, K. E. (1956). *The Image.* Ann Arbor, MI: University of Michigan.

Brobst, S. A., Malone, T. W., Grant, K. R. and Cohen, M. D. (1986). Toward intelligent message routing systems. In *Computer message systems-85: Proceedings of the Second International Conference on Computer Message Systems* (pp. 351-359). Amsterdam: North-Holland.

Carroll, L. (1960). *The Annotated Alice: Alice's Adventures in Wonderland & Through the Looking Glass.* New York: Bramhall House. (Original works published in 1865 and 1871.)

Chellas, B. F. (1980). *Modal Logic: An Introduction.* Cambridge: Cambridge University Press.

Child, J., Bertholle, L. and Beck, S. (1981). *Mastering the Art of French Cooking.* New York: Alfred A. Knopf.

Civil aerospace: The show goes on. (1988, September 3). *Economist*, pp. Survey 5–28.

Clark, K. B. (1989). Project scope and project performance: Effect of parts strategy and supplier involvement on product development. *Management Science, 35*(10), 1247–1263.

Cohen, P. R. and Perrault, C. R. (1979). Elements of a plan-based theory of speech acts. *Cognitive Science, 3*, 177–212.

Covert, E. (1989). *The commercial aircraft industry: Overview.* Presentation for "America's competitive challenge in the 90s", October 2, Cambridge, MA: MIT Sloan School.

Coxe, W. (1980). *Managing architectural and engineering practice.*

Crowston, K. and Treacy, M. E. (1986). Assessing the impact of information technology on enterprise level performance. In *Proceedings of the Sixth International Conference on Information Systems* (pp. 299-310). Indianapolis, IN.

Crowston, K., Malone, T. W. and Lin, F. (1987). Cognitive science and organizational design: A case study of computer conferencing. *Human Computer Interaction, 3*, 59-85.

Cyert, R. M. and March, J. G. (1963). *A Behavioral Theory of the Firm.* Englewood Cliffs, NJ: Prentice-Hall.

Davis, E. (1990). *Representations of Commonsense Knowledge.* San Mateo, CA: Morgan Kaufmann.

Davis, R. and Smith, R. G. (1983). Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence, 20*, 63–109.

Debreu, G. (1959). *Theory of value: An axiomatic analysis of economic equilibrium.* New York: Wiley.

Decker, K. S. and Lesser, V. R. (1989). Some initial thoughts on a generic architecture of CDPS network control. In M. Benda (Ed.), *Proceedings of the*

Ninth Workshop on Distributed Artificial Intelligence (pp. 73–94). Rosario Resort, Eastsound, WA.

Dennett, D. C. (1987). *The Intentional Stance.* Cambridge, MA: MIT Press.

Durfee, E. D. and Lesser, V. R. (1987). Using partial global plans to coordinate distributed problem solvers. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)* (pp. 875–883).

Durfee, E. H. (1988). *Coordination of Distributed Problem Solvers.* Boston, MA: Kluwer Academic.

Eco, U. (1989). *Focault's Pendulum.* New York: Ballantine.

Embry, J. D. and Keenan, J. (1983). Organizational approaches used to improve the quality of a complex software product. In R. S. Arnold (Ed.), *Software Maintenance Workshop* (pp. 131–133). Monterey, CA: Naval Postgraduate School.

Emerson, R. M. (1962). Power-dependence relationships. *American Sociological Review, 27,* 31–40.

Emery, J. C. (1969). *Organizational Planning and Control Systems: Theory and Technology.* New York: MacMillan.

Engineering made easy, thanks to Igor and friends. (1990, September 3). *Business Week,* pp. 103–104.

Erman, L. D., Hayes-Roth, F., Lesser, V. R. and Reddy, D. R. (1980). The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys, 12*(2), 213–253.

Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence,* 2, 198–208.

Fitzgerald, K. (1989). Probing Boeing's crossed connections. *IEEE Spectrum, 26*(5), 30–35.

Forget miles-per-gallon. (1988, January 30). *Economist*, pp. 50–51.

Fox, M. S. (1981). An organizational view of distributed systems. *IEEE Transactions on Systems, Man and Cybernetics, 11*(1), 70–79.

Franks, N. R. (1989). Army ants: A collective intelligence. *American Scientist, 77*(March-April), 139–145.

Freeman, L., Romney, A. K. and Freeman, S. E. (1987). Cognitive structure and informant accuracy. *American Anthropologist, 89*, 310–325.

Galbraith, J. R. (1974). Organization design: An information-processing view. *Interfaces*, 4(5), 28-36.

Galbraith, J. R. (1977). *Organization Design*. Reading, MA: Addison-Wesley.

Gasser, L. and Huhns, M. N. (Eds.). (1989). *Distributed Artificial Intelligence* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

Gasser, L. and Rouquette, N. (1988). Representing and using organizational knowledge in distributed AI systems. In *1988 Workshop on Distributed Artificial Intelligence*. Lake Arrowhead, CA.

Glaser, B. G. and Strauss, A. L. (1967). *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Chicago: Aldine Publishing.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The language and its implementation*. Reading, MA: Addison-Wesley.

Gresov, C. (1988). *Exploring Fit and Misfit with Multiple Contingencies* (Unpublished paper. Durham, NC: Fuqua School of Business, Duke University.

Hackman, J. R. (1969). Toward understanding the role of tasks in behavioural research. *Acta Psychologica, 31*, 97–128.

Halpern, J. Y. and Moses, Y. (1985). A guide to the modal logics of knowledge and belief: Preliminary draft. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)* (pp. 480–490).

Hayes, P. J. (1977). In defence of logic. In *Proceedings of the Fifth International Joint Conference on AI (ICJAI-77)* (pp. 559–565). Cambridge, MA: Available from the Department of Computer Science, CMU.

Henderson, R. M. and Clark, K. B. (1989). *"Generational" innovation: The reconfiguration of existing systems and the failure of established firms* (Working Paper 3027-89-BPS). Cambridge, MA: MIT Sloan School.

Hewitt, C. (1986). Offices are open systems. *ACM Transactions on Office Systems, 4*(3), 271–287.

Hilton, J. R. (1985). *Design Engineering Project Management: A Reference.* Lancaster, Pa.: Technomic Pub. Co.

Holmstrom, B. (1979). Moral hazard and observability. *Bell Journal of Economics, 10*(Spring), 74–91.

Huberman, B. A. (Eds.). (1988). *The Ecology of Computation.* Amsterdam: North-Holland.

Huhns, M. (Eds.). (1987). *Distributed Artificial Intelligence.* San Mateo: Morgan Kaufmann.

Huhns, M. N. and Gasser, L. (Eds.). (1989). *Distributed Artificial Intelligence* (Vol. 3). San Mateo, CA: Morgan Kaufmann.

Jensen, M. C. and Meckling, W. H. (1979). Rights and production functions: An application to labour managed firms and codetermination. *Journal of Business, 52*(4), 469–506.

Kidder, L. H. (1981). *Research Methods in Social Relations* (4th ed.). New York: Holt, Rinehart and Winston.

Lai, K. Y., Malone, T. and Yu, K.-C. (1988). Object Lens: A spreadsheet for cooperative work. *ACM Transactions on Office Information Systems, 6*(4), 332–353.

Leech, D. J. (1972). *Management of Engineering Design.* New York: J. Wiley.

Lientz, B. P. and Swanson, E. B. (1980). *Software Maintenance Management: A Study of the Maintenance of Computer Applications Software in 487 Data Processing Organizations.* Reading, MA: Addison-Wesley.

Lochbaum, K. E., Grosz, B. J. and Sidner, C. L. (1990). Models of plans to support communications: An initial report. In *Proceedings of the 8th National Conference on Artificial Intelligence* (pp. 485–490). Cambridge, MA: MIT Press.

Malone, T. W. (1987). Modeling coordination in organizations and markets. *Management Science, 33,* 1317–1332.

Malone, T. W. (1988a). Modeling coordination in organizations and markets. In A. H. Bond and L. Gasser (Ed.), *Readings in Distributed Artificial Intelligence* (pp. 151–158). San Mateo, CA: Morgan Kaufman.

Malone, T. W. (1988b). *What is coordination theory?* (Working paper #2051-88). Cambridge, MA: MIT Sloan School of Management.

Malone, T. W. and Crowston, K. (1990). What is coordination theory and how can it help design cooperative work systems? In D. Tatar (Ed.), *Proceeding of the Third Conference on Computer Supported Cooperative Work* (pp. 357–370). Los Angeles, CA: ACM Press.

Malone, T. W., Grant, K. R., Turbak, F. A., Brobst, S. A. and Cohen, M. D. (1987). Intelligent information-sharing systems. *Communications of the ACM, 30,* 390–402.

Marca, D. A. and McGowan, C. L. (1988). *SADT™: Structured Analysis and Design Technique.* New York: McGraw-Hill.

March, J. G. and Simon, H. A. (1958). *Organizations.* New York, NY: John Wiley and Sons.

Marschak, J. (1955). Elements for a theory of teams. *Management Science, 1,* 127–137.

Marschak, J. and Radner, R. (1954). The firm as team (abstract). *Econometrica, 22,* 523.

McCann, J. E. and Ferry, D. L. (1979). An approach for assessing and managing inter-unit interdependence. *Academy of Management Review, 4*(1), 113–119.

McCann, J. E. and Galbraith, J. R. (1981). Interdepartmental relations. In P. C. Nystrom and W. H. Starbuck (Ed.), *The Handbook of Organizational Design* (Vol. 2) (pp. 60–84). New York: Oxford University Press.

McDermott, D. (1978). Tarskian semantics, or No notation without denotation! *Cognive Science, 2,* 277–282.

McGrath, J. E. (1984). *Groups: Interaction and Performance.* Englewood Cliffs, NJ: Prentice-Hall.

McKenney, J. L., Doherty, V. S. and Sviokal, J. J. (1986). *The impact of electronic networks on management communications: An information processing study* (Working Paper 1-786-041). Boston, MA: Harvard Business School.

Miller, M. S. and Drexler, K. E. (1988). Markets and computation: Agoric open systems. In B. A. Huberman (Ed.), *The Ecology of Computation* (pp. 133–176). Amsterdam: North-Holland.

Minsky, M. (1987). *The Society of the Mind.* New York: Simon and Schuster.

Mintzberg, H. (1979). *The Structuring of Organizations.* Englewood Cliffs, NJ: Prentice-Hall.

Moore, R. C. (1979). *Reasoning About Knowledge and Action.* Unpublished Ph. D. thesis, Department of Electric Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Moore, R. C. (1982). The role of logic in knowledge representation and commonsense reasoning. In *Proceedings of AAAI National Conference on AI* (pp. 428–433). Pittsburgh, PA: AAAI.

Morgenstern, L. (1988). *Foundations of a Logic of Knowledge, Action and Communication*. Unpublished Ph. D. Dissertation, New York University, Department of Computer Science, New York, NY.

Newell, A. (1979). Reasoning, problem solving and decision processes: The problem space as a fundamental category. In R. Nickerson (Ed.), *Attention and Performance*. Erlbaum.

Newell, A. and Simon, H. A. (1972). *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall, Inc.

Newhouse, J. (1983). *The Sporty Game*. New York: Alfred A. Knopf.

Pollack, M. (forthcoming). Plans as complex mental attitudes. In P. Cohen (Ed.), *Intentions in Communications*. Cambridge, MA: MIT Press.

Prietula, M. J., Beauclair, R. A. and Lerch, F. J. (1990). A computation view of group problem solving. In *Proceedings of the 23rd Hawaii International Conference on System Sciences*. Kailua-Kona, Hawaii: IEEE Computer Society Press.

R & D scoreboard: A perilous cutback in research spending. (1988, June 20). *Business Week*, pp. 139–162.

Roberts, K. H. (1990). Some characteristics of one type of high reliability organizations. *Organization Science, 1*(2), 160–176.

Rochlin, G. I., Porte, T. R. L. and Roberts, K. H. (1987). The self-designing high-reliability organization: Aircraft carrier flight operations at sea. *Naval War College Review, 60*(4, Autumn), 76–90.

Ross, S. (1973). The economic theory of agency. *American Economic Review, 63*, 134–139.

Schank, R. C. and Abelson, R. P. (1977). *Scripts, Plans, Goals and Understanding: An Inquiry into Human Knowledge.* Hillsdale, NJ: Lawerence Erlbaum Associates.

Searle, J. R. (1969). *Speech Acts: An Essay in the Philosophy of Language.* Cambridge: Cambridge University.

Seeley, T. D. (1989). The honey bee colony as a superorganism. *American Scientist, 77*(November-December), 546–553.

Simpson, J., Field, L. and Garvin, D. A. (1988). *The Boeing 767: From Concept to Production (A)* (Case 9-688-040). Boston, MA: Harvard Business School.

The sincerest form of flattery. (1989, November 11). *Economist*, pp. 79–82.

Smith, R. G. and Davis, R. (1981). Frameworks for cooperation in distributed problem solving. *IEEE Transactions on Systems, Man and Cybernetics, 11*(1), 61–70.

Star, S. L. (1989). The structure of ill-structured solutions: Boundary objects and heterogeneous distributed problem solving. In L. Gasser and M. N. Huhns (Ed.), *Distributed Artificial Intelligence* (Vol. 2) (pp. 37–54). San Mateo, CA: Morgan Kaufmann.

Stefik, M. and Bobrow, D. G. (1986). Object-oriented programming: Themes and variations. *AI Magazine*, (Spring), 40–62.

Stuart, C. (1985). An implementation of a multi-agent plan synchronizer. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)* (pp. 1031-1033).

Taking the strain. (1989, January 28). *Economist*, pp. 68.

Tenenberg, J. D. (1988). *Abstraction in planning* (Technical Report 250). Rochester, NY: Computer Science Department, University of Rochester.

Thompson, J. D. (1967). *Organizations in Action: Social Science Bases of Administrative Theory.* New York: McGraw-Hill.

Thoreau, H. D. (1958). *Walden or, Life in the Woods and On the Duty of Civil Disobedience.* New York: Harper & Row. (Original work published in 1854.)

Trying times at Boeing. (1989, March 13). *Business Week*, pp. 34–36.

Tushman, M. and Nadler, D. (1978). Information processing as an integrating concept in organization design. *Academy of Management Review, 3*, 613–624.

Van de Ven, A. H., Delbecq, A. L. and Koenig, R., Jr. (1976). Determinants of coordination modes within organizations. *American Sociological Review, 41*(April), 322–338.

Victor, B. and Blackburn, R. S. (1987). Interdependence: An alternative conceptualization. *Academy of Management Review, 12*(3), 486–498.

von Bertalanffy, L. (1950). The theory of open systems in physics and biology. *Science, 111*.

von Martial, F. (1989). Multiagent plan relationships. In M. Benda (Ed.), *Proceedings of the Ninth Workshop on Distributed Artificial Intelligence* (pp. 59–72). Rosario Resort, Eastsound, WA.

Wiener, N. (1961). *Cybernetics: Or Control and Communication in the Animal and the Machine* (2nd ed.). Cambridge, MA: MIT Press.

Wilensky, R. (1983). *Planning and Understanding: A Computational Approach to Human Reasoning.* Reading, MA: Addison-Wesley.

Winograd, T. (1972). *Understanding Natural Language.* New York: Academic Press.

Yin, R. K. (1984). *Case study research: Design and methods.* Beverly Hills, CA: Sage.

Yourdon, E. (1989). *Modern Structured Analysis.* Englewood Cliffs, NJ: Yourdon.