THE MENTOR PROJECT:
REPLICATING THE FUNCTIONING OF AN ORGANIZATION

H. Joel Jeffrey and Anthony O. Putman

ABSTRACT

The MENTOR project addresses the problem of modeling a large, complex human organization in sufficient detail for the model to be practically useful. In this project the process of developing software in an organization of over 300 people was studied. The project includes building a model of the software development process in the organization and writing a program which uses the model to furnish accurate, specific, current information needed by software developers. This information system has been implemented. It accepts a developer's question in ordinary technical English, uses the model to figure out an answer tailored to the specific developer and the circumstances, and answers the developer in English. A pilot study was carried out, in which the developers found the system to be accurate, informative, and useful. The project is based on an unusual conceptualization, the human system approach. Descriptive Psychology provides the theoretical and technical basis for acting on this conceptualization. This chapter presents the human system approach, the application of Descriptive Psychology to the problem, and the pilot project in detail.

Advances in Descriptive Psychology, Volume 3, pages 243–269
Editors: Keith E. Davis and Raymond M. Bergner

ISBN 0-89232-293-4

The MENTOR project is an attempt to solve a previously intractable problem: How can we model, at sufficient depth of complexity and detail to be useful for practical purposes, a large, complex organization, including the people and their activities and computer software and its activities? The organization modeled is the Toll Digital Switching Laboratory of Bell Telephone Laboratories. Bell Telephone Laboratories is the research and development organization of American Telephone and Telegraph, Inc.; the Toll Digital Switching Laboratory, which includes about 340 people, is responsible for building and maintaining the long-distance telephone switching system known as No. 4 ESS (Electronic Switching System). The particular area of activity within that organization that we chose as the area of investigation was the process of developing switching system software.

The project is based on a significantly different conceptualization, which we call the "human system" approach. This paper presents the approach, the way we followed through on it and describes in detail the pilot study that was done to test the feasibility of the project. The pilot study included modeling a sizeable portion of the software development process in the Laboratory, writing a program to access that model to answer actual user questions, and trial and evaluation of the system by a significant body of users in the Laboratory.

The process of developing software for No. 4 ESS is large and complex. Although the process is generally stable, its details change due to varying technical needs, sufficiently often that software developers report significant difficulty in keeping their information current. That information includes a great wealth of knowledge about people, activities, and interactions between those activities: writing code, debugging one's code, integration testing, developing a new feature, reporting problems, telling others what you've done, filling out Failure Reports (FRs) and Correction Reports (CRs), processing those reports, etc. A software developer whose knowledge is too far out of date is significantly restricted in his ability to engage in the practices involving the knowledge; this is particularly acute when many of the practices involve using other computer programs, which requires having very small details exactly right.

As is often the case in large organizations, the generla level of understanding of the software development process is (in the opinion of experts in the Laboratory) not what one might hope for. Because of its size and complexity, and the large number of people who are either new or have been fairly sharply focused in one area, Laboratory personnel generally understand the process well enough to get the daily job done, but their understanding tends to be superficial or narrow. When unusual cases or circumstances arise, as they inevitably do, developers quickly find themselves having to do things they are unfamiliar with. The level of understanding is low enough to have a significant negative impact on the productivity of the developers.

In addition to the difficulty of keeping up to date, experts in the Laboratory consider it to be difficult to acquire expertise in the development process, both for new people and those moving into a new area. It seems to take a new person

approximately two full cycles of developing a new version of the No. 4 software (about three years) to become really competent in the software development process.

Documentation of the development process is lacking for two reasons. First, a large portion of the knowledge is knowledge about a complex, interrelated set of activities done by people, that is, a complex set of human behaviors. There is little disagreement that what people need is a unifying framework for this knowledge, rather than simply facts or memos; this wealth of data about people and human behavior has, until now, proven extremely resistant to all attempts to gather it and build a useful model of it.

Second, would-be documenters have been faced with a dilemma. Since the development process *is* large and complex, any documentation of it that is both complete and detailed enough to be useful in the daily activities of developers would be so massive as to be virtually unusable.

The MENTOR project is an attempt to deal with this need for information by (a) gathering the information various developers have about how things really work, (b) building a model of the development process, and (c) using that model as the basis for a computerized system that developers can use to get answers to their questions about the process.

The point of doing this is to reduce the number of mistakes by developers, and to increase the developers' effectiveness in the organization by supplying missing, but necessary, information. If an individual who lacks some information about a computer program makes a change in it, he or she is very likely to make an error. In an analogous manner, someone missing some information about the software development process will quite probably either have to take the time to find the information, or not take the time and make an error. The cost in time and effort, by the developer and others, is significant.

In addition to the immediate impact of these errors and confusion, there is the cost of one of any organization's most precious resources: the time of the comparatively few persons who are experts on how to get things done. Several such experts in the Laboratory reported that they had in the past spent a good deal of time helping developers needing information and repairing damage done by developers who had not had the proper information (or, in some cases, had not known that there was information to get). The experts in how to get things done are almost always those who have been in the organization longest, and it would generally be preferable if they did not need to spend so much of their time in this manner, because they are the ones with the most to contribute to difficult technical problems, including the design of innovations.

All of this adds up to extra staff effort on development projects, extra time, and confusion and annoyance for members of the organization—in this case, people developing No. 4 ESS software.

Successfully meeting the software developers' need for accurate, current, easily used information would have two important benefits. First, it would enable developers to do things more quickly and easily, by making it easy to find

out what they need or want to know in order to act. Second, it would have a significant impact on morale of the organization, since it is very difficult for people to experience satisfaction in a confusing, annoying environment. (It seems quite probable to us that the effect on morale would in turn have a further impact on the productivity of the organization.)

The first step was to do a pilot project to test the approach and the technology. This was the first time this sort of application had been attempted. The pilot project was designed to address the following questions:

1. Could we in fact describe the software development process completely and at a technically useful level of detail?
2. Could we in fact use the description to give people information in a useful form? (This question very soon became, Could We write a program, with a reasonable amount of time and effort, to answer people's real questions?)
3. Assuming a positive answer to the above questions, would people in the Laboratory actually use the tool?

The approach and the technology of the project are described first, then the pilot study is presented, and finally current efforts and extensions are discussed.

HUMAN LOGIC AND PROCESS DESCRIPTIONS

There are two key differences between the MENTOR project and other approaches to modeling software development: the basic approach, or perspective, and the technical basis for following through on that approach.

Human Systems

The basis of the MENTOR project is to view the software development process as a ***human system,*** that is, as an interrelated set of activities done by people. Accordingly, the system is best described with concepts and a technology suitable for human systems, rather than some other sort. This does not mean that the usual facts, objects, and processes are ignored, but rather that another set of facts, objects, and processes will be included—facts and processess including humans.

The human system approach contrasts fairly sharply with more typical approaches. The most frequent technical approach to the problem is to build some type of finite-state machine model. Directed graphs, flowcharts, decision table, cause-and-effect methodology, or finite-state machines are all examples. This approach has run into two difficulties.

First, the technology is ill-suited to describing a human system, since it is not designed to describe specifically human behavior (we shall see an example of this shortly). As a result, attempts to use the finite-state machine model quickly result in very complex graphs (or tables, etc.), which are not only very difficult to use, but which, to our knowledge, have not been successfully used as a basis for the kind of information system that would meet the needs addressed by this project.

Second, other approaches that we are aware of are designed, intentionally, to represent abstractions of what actually happens. However, a member of an organization with a job to do, who needs a question answered in order to do the job, wants an *answer,* not something he or she can (assuming the relevant competence) use to figure out an answer. In other words, an abstraction is not likely to be directly usable by a person asking a question. Therefore, the problem of translating the abstraction into what actually happens in the organization remains.

In order to see the kind of difficulty one must deal with in describing human systems, consider the following example:

Two people, Gil and Wil, observe Jill sitting at a computer terminal, punching the keys. Gil asks Wil, "What is Jill doing?" Wil, depending on his preference and personal characteristics, could give any of the following answers:

1. Editing a file
2. Changing the source code of a program
3. Fixing a bug in a program
4. Maintaining the current release of the software

Here we have a problem. Each of these answers is a useful, informative description; each names an activity that actually happens. Further, each description has direct connections to very different other activities. (For example, the next step in the activity named "editing a file" is different from the next step in "fixing a bug".) Which, then, shall we say that Jill is doing? Further, if we are to write a program that can answer Jill's questions about, for example, what to do next, how will that system tell which description is "right"?

The solution to the problem, of course, is not to take the bait. What is actually happening is that

1. Jill is editing a file; by doing this,
2. Jill is changing the source code of a program; by doing this,
3. Jill is fixing a bug in a program; by doing this,
4. Jill is maintaining the release of the software.

In other words, Jill is engaging in *all* of ~~these~~ activities, *simultaneously*.

Further, as one might guess from the mundane character of this example, doing more than one thing simultaneously, rather than being unusual, is by far the most common case in human behavior. Therefore, any model of a system involving people that does not include this hierarchical, simultaneous character of human behavior will be leaving out some important facts and connections to other activities.

A primary way in which we have acted on our view of the software development process as a human system has been to develop and use a set of descriptions of the process that are statements of what persons in the Laboratory do, and the ways those activities interact. Thus, we have built a model of the process as a *human* system. This model is the needed "blueprint" of what actually happens in developing software in the Laboratory, and forms the critical, but up until now missing, unifying framework for the facts and data.

Descriptive Psychology

It was clear from the outset of the project that if the human system approach was to be more than simply a slogan or an abstract philosophy, one element was absolutely necessary: a theory that was

1. Designed for describing human (as opposed to chemical, physical, finite-state, etc.) systems.

2. Applicable to the problem of representing what actually happens as people develop software. Theories of emotion, self-actualization, politics, and so on might or might not be useful, but would not address the developers' need for information.

3. Technically usable, that is, one that would allow us to give descriptions that we could work with technically. No matter how good the descriptions were, it seemed very unlikely that *any* set of ordinary English descriptions would allow us to give developers or managers directly usable information.

Descriptive Psychology (Ossorio, 1969/1978; 1971–1978b) was the only conceptual framework known to the authors that met these criteria, and so was the natural choice for the technical basis for the project. The project relies technically on giving descriptions of the software development process using the basic process unit, or BPU (Ossorio, 1971/1978b). Since using the basic process unit is the technical foundation of the project, the next section discusses it in detail.

Describing Processes

The basic process unit is the primary device for reaching a technically effective level of detail in describing a process (Ossorio, 1971/1978b, pp. 40–41). It

is a means of codifying all of the information about a process, at a given level of detail.

Rather than attempt to restate the explanation of the BPU (Ossorio, 1971/1978b), it seems more useful to illustrate its use, with some actual information about the software development process we are modelling. We will first present the information discursively, and then show how it is codified in a basic process unit. (In a few instances we will use some actual technical names without defining them.)

Among the activities involved in developing and maintaining a No. 4 ESS Generic (a version of the software and hardware) is the activity of finding and fixing problems. The next paragraphs are a brief description of part of what is involved in one step of this activity: reporting a problem.

The first step in finding and fixing a problem in No. 4 ESS is that some responsible person in the Laboratory finds out about it. This can happen in four ways: (a) Someone at Indian Hill (the location of the Laboratory) discovers a problem and reports it by filing a Failure Report (FR) on it. (b) Someone at Indian Hill finds a problem and tells the responsible programmer (the programmer responsible for the part of the software at fault). (c) Someone at Indian Hill finds the problem and tells the Group FR Coordinator of the Group responsible for the program needing the fix about it. The Group FR Coordinator either files the FR himself, or has the programmer file it. (d) Someone in a Field Office (a telephone office where No. 4 ESS is installed) finds a problem and reports it to someone in the No. 4 Electronic Switching Assistance Center (4ESAC) at Indian Hill. Someone in 4ESAC then files an FR on the problem, or contacts someone in the Field Support Group about it. The Field Support person then either files the FR, contacts the responsible programmer about the problem, or has the Group FR Coordinator for the Group responsible for the program needing the fix file the FR.

Anyone at Indian Hill can find and report the problem. The Group FR Coordinator will be the FR coordinator for one of the Development Groups in the Laboratory (Call processing, Data Administration, MAS & INWATS, Network Management, and Trunk Maintenance).

Certain points about this information are worth noting. First, it is entirely concerned with activities done by people. While technical elements are present (No. 4 ESS, the fix for the problem, etc.), we are talking about human actions here. Second, information about the ways people can do some task, the steps involved in it, which people do which parts, and so on, is crucial to developers, for understanding their work and for doing their jobs effectively. Finally, this information does not exist, in written form, anywhere. It is part of what "everybody knows", and is passed entirely by word of mouth. Of course, a great many people do *not* know it, especially those new to the job.

Now let us codify this information in a basic process unit. The BPU consists of a pair, the Name and the Description. The Name identifies the process, and is

used to refer to it. The Description contains the details of the process. As we shall see, both the Name and the Description are formal items, not English text. To identify the process, we give it a name that we, the describers, judge useful and appropriate. ''Name 1'', ''open data file for update'', ''throw a ball'', ''fire a gun'', and ''fix a bug'' are all examples of Names.

The person writing the BPU has free choice of Names. The basis for choosing is the informative value for the person (or persons) who will be reading the descriptions. In light of this, the describer typically chooses Names that are a brief description of the process, as that is typically the easiest way to identify the process to the reader. The Name of the process discussed about is ''responsible persons in the Laboratory find out about a problem''.

As we noted, finding out about a problem is part of the larger activity of finding and fixing a problem in No. 4 ESS; this larger activity is a process with the Name, ''responsible persons find and fix a problem in a No. 4 Generic''.

Sometimes a Name is all that is needed for the use the BPU will be put to; sometimes not. To give the details of a process, one gives the Description. This Description has six parts: Stages, Elements, Individuals, Eligibilities, Contingencies, and Versions.

Stages. A process breaks down into Stages, which may be sequential or parallel. The stages constitute a task analysis of the process; they are the subprocesses of the process being described. Stages are specified by Names; thus, the basic information on how a process is done is given by *other process Names.*

This is an important difference between the basic process unit and other approaches to describing complicated sets of activities, which sometimes follow the Name-Description format. With other approaches, the Description is typically in some form of English text (although it may be a somewhat structured form of English). The breakdown of a process with a purely formal name into other purely formally named processes is critical to the concept of using the basic process unit to describe a process. This formalism is what makes it possible to write a program to work with a set of process descriptions without the problems of natural language understanding and artificial intelligence.

The Stages of ''find and fix a problem'' are:

1. Responsible persons in the Laboratory find out about a problem
2. People who keep track of problems track the course of the problem
3. The responsible programmer decides the response to the problem
4. The responsible programmer implements the chosen response to the problem
5. People in a support group install the fix for the problem in the Generic.

Each stage, in general, may be accomplished in more than one way. In our illustration, there are four ways in which a responsible person in the Laboratory can find out about a problem. These ways of doing the Stage are the *Options*. In "find and fix a problem," the Options for Stage 1 are as follows:

1. Responsible persons in the Laboratory find out about a problem
   - 1-1 a person at Indian Hill discovers a problem and reports it
   - 1-2 a person at Indian Hill discovers a problem and has the responsible programmer file the FR on it
   - 1-3 a person at Indian Hill discovers a problem and has the FR Coordinator tell the programmer about it
   - 1-4 a person in a No. 4 Field Office discovers a problem and reports it.

The options for Stage 4 are as follows:

4. The responsible programmer implements the chosen response to the problem
   - 4-1 the responsible programmer produces the fix for the problem
   - 4-2 the responsible programmer files a Not-applicable CR (a Correction Report stating that the problem is not applicable to this version of the software) for the problem
   - 4-3 the responsible programmer files a Not-implemented CR (a Correction Report stating that the problem is something that has not yet been implemented in the software) for the problem
   - 4-4 the responsible programmer files a Cancel CR (a Correction Report stating that the problem report has been cancelled).

Notice that the four Options for Stage 1 are the four ways of finding out about a problem presented in the discursive description given earlier.

The Stages and Options codify the subprocess structure of the process. Other aspects remain, however: the state-of-affairs structure of the process. To codify this structure one uses the Elements, Individuals, Eligibilities, and Contingencies (Ossorio, 1971–1978b, p. 43). The last paragraph of the discursive description of reporting a problem provides this sort of information.

Elements. We normally consider a process to have object constituents. The Elements are these objects. The Elements in a BPU are the objects appearing in the Names of the process, Stages, or Options. Again, they are specified by purely formal names, chosen by the describer to be informative to the reader. Some of the Elements in the BPU we have so far are:

1. person at Indian Hill
2. problem
3. fix for the problem
4. response to the problem
5. NA CR
6. NI CR
7. CC CR
8. No. 4 Generic
9. FR Coordinator
10. responsible programmer
11. people who keep track of problems.

Individuals. In an actual occurrence of a process one sees actual historical individuals taking the places named by the Elements. For example, "No. 4 Generic" may (today) be any of the individuals named 4E4, 4E5, or 4E6. These Individuals play the roles named by the Elements. Again, the Individuals are given by formal names, chosen to be informative. In our example, a partial list of the Individuals is:

1. anyone at Indian Hill
2. member of 4ESAC
3. 4E4
4. 4E5
5. 4E6
6. persons in a support group
7. the FR for the problem
8. the CR for the problem
9. incorrect behavior by the ESS machine
10. people in System Test
11. people in a development group
12. people in the Field Support Group
13. people in a development Department.

Eligibilities. The specification of the Eligibilities of Individuals to be Elements codifies the information about which historical individuals may take which parts in the process. Examples of Eligibilities are:

1. the No. 4 Generic may be 4E4, 4E5, or 4E6
2. person at Indian Hill may be any technical staff member at Indian Hill
3. people who keep track of problems may be people in System Test, people in a development Group, people in the Field Support Group, or people in a development Department.

Contingencies. Not all of the Stage-Option combinations are valid instances of the process, that is, would not be considered instances of the process being described. Valid combinations are specified by certain of the Contingencies which specify which Stages and Options are sequential or parallel. Contingencies are of two types: *Attributional Constraints* and *Co-occurrence Constraints*.

An Attributional Constraint specifies that the occurrence of an Option within a Stage, or an entire Stage, depends on some Element's having an Attribute. For example, Stage 1—Option 4 (a Field Office reporting a problem) happens if and only if the Generic is in the field. (It could be that the Generic is under development.)

Co-occurrence Constraints specify that certain Stages, or Options of a Stage, take place only if other Stages or Options do. For example, the programmer produces the fix for the problem (Stage 4—Option 1) only if, in choosing the response to the problem, the programmer chooses the Option "decide to fix the problem"—Option 1 of Stage 3 of the process named "responsible programmer chooses the response to the problem."

The Constraints are encoded formally, so that they can easily be processed by a program; Attributional Constraints are of the form

Stage-Option Name if Attribute

where the Attribute, again, is a purely formal name. Co-occurrence Constraints are of the form

Stage-option Name code Stage-Option Name

where "code" is only-after, only-before, or if-and-only-if.

This formal structuring of the information makes it easy for a program to search for things that must be done before some action—prerequisite constraints. For example, if a person asks how to file a CR, a program can find all processes in which this is a Stage or Option, search the list of Constraints in each of those processes for actions that must be done before "file a CR," and then check whether those actions have been taken, if necessary by querying the user.

Versions. Actual instances of the process being described are known as Versions. Each Version is a sequence of Stage-Option Names satisfying the Constraints. In our example, some of the Versions are:

1. 1–1, 2, 3, 4–1, 5–1
2. 1–2, 2, 3, 4–1, 5–1
3. 1–3, 2, 3, 4–1, 5–1
4. 1–4, 2, 3, 4–1, 5–1
5. 1–1, 2, 3, 4–2, 5–1

(Single numbers refer to Stages, e.g., 2 is "keeping track of the problem"; hyphenated numbers refer to Options within a stage, for example, 4-1 refers to "fixing the problem".)

We have found it useful to distinguish certain types of Versions, including paradigm Versions, preferred Versions, and error Versions. In the above list, the first Version is a paradigm case of this process: Someone at Indian Hill discovers the problem and reports it, people track the handling of the problem, the programmer decides what to do about the problem, the programmer produces a fix for the problem, and people in a support group install the fix. The preferred Versions are the preferred ways to do the process, if one is Eligible. For example, if one is authorized, it is preferred to sign one's own CR. Finally, the error Versions are ways in which the process can actually be done, but which it should not be. For example, it is possible to take an FR number but not sign it out in the Log meant for that purpose. Doing this causes difficulties for the person who does it or for others. It is not supposed to happen, but does. An example of Error Version in our example would be for a person to discover and report the problem (Stage 1—Option 1), and for people to track it (Stage 2), but for the responsible programmer to ignore it when he is notified (error in Stage 3).

THE PILOT PROJECT

Several researchers have used Descriptive Psychology in technical applications (Jeffrey, 1975; Johannes, 1977; Ossorio, 1966; 1971/1978a), including describing a human system of significant size and complexity (Busch, 1974). No one, though, had used it to both build a model of an actual human system and to write a software package to manipulate the model. As we noted earlier, a pilot project was carried out to address the issues of (a) the descriptive adequacy of the approach and the basic process unit, (b) the possibility of using the model to meet the developers' needs for information, and (c) the judgment by the developers themselves of the usefulness of the information system. We selected a portion of the development process that was sufficiently complex to test the approach, and of sufficient interest in its own right that the resulting information system would be of interest to developers. We wrote a program to answer questions developers might typically ask, and then had a sample of developers in the Laboratory use and evaluate the program.

Subject Matter

Within all of the practices involved in building and maintaining No. 4 ESS, there is a set of activities involving finding bugs, reporting them, fixing them, and getting the fixes into the software. These activities are commonly referred to in the Laboratory as "FR/CR handling". This is the subject matter covered for the pilot project. It ranges from high level descriptions, such as "find and fix a problem", down to the most concrete aspects of the development process, such as which priority code to put in the priority code box on a form.

Covering this material required about 50 basic process unit descriptions, with 399 Stage-Option names. Six levels were encompassed from the highest level Name (''find and fix a problem in a Generic'') to the lowest (''filling out special instructions on a Correction Report'').

This area was chosen primarily because it was suggested by a number of developers as a very complex subject about which there was a great deal of confusion. Further, FR/CR handling includes many activities done only by people: reporting problems, giving forms to other people, checking up on the handling of a problem, and so on, as well as filling out forms. Thus, it was considered a good test of the adequacy of the descriptive technique, especially for those parts of the software development process that include specifically human activities.

Here are some examples of basic process unit descriptions (including only the Name and the Stage-Options):

NAME: responsible people find and fix a problem in a No. 4 Generic
Stages:

- a responsible person at Indian Hill finds out about problem
  - Option: person at Indian Hill discovers a problem and reports it
  - Option: person at Indian Hill discovers a problem and has the responsible programmer file the FR on it
  - Option: person at Indian Hill discovers a problem and has the FR Coordinator tell the programmer about it
  - Option: a person in a No. 4 Field Office discovers a problem and has someone in 4ESAC tell the programmer about it
- people who keep track of problems track the course of the problem
- the responsible programmer decides the response to the problem
- the responsible programmer implements the response to the problem
  - Option: the responsible programmer fixes the problem
  - Option: the responsible programmer files an NA CR for the problem
  - Option: the responsible programmer files an NI CR for the problem
  - Option: the responsible programmer files a CC CR for the problem
- people in a support group install the fix for problem in No. 4 Generic
  - Option: people in System Test install the fix for the problem in the Generic under development
  - Option: people in the Field Support Group install the fix for the problem in a field Generic

NAME: person files an FR on problem
Stages:

- person gets an FR form
  - Option: person gets an FR form from the System Lab
  - Option: person gets a previously signed-out FR form

- person fills out the FR form
- person gets the Supervisor's signature on FR form
  - Option: Supervisor signs FR form
  - Option: a person who can sign his/her own FR signs Supervisor's signature on FR form
- person submits FR form
  - Option: person puts the FR in the System Lab bin
  - Option: person working on Recent Change and Verify puts the FR in the box in the Western Electric Recent Change Group area
  - Option: person give the FR to Field Support Group FR Coordinator
  - Option: person gives the FR to Dept. 5426 FR Coordinator
  - Option: person working on Recent Change and Verify gives the FR to Data Administration Group FR Coordinator

NAME: the responsible programmer fixes the problem

Stages:

- the responsible programmer produces code to fix the problem
- the responsible programmer files a PIDENT change CR
- the responsible programmer PERMs the overwrite on TSS
- TSS executes the PERMOW procedure
- the responsible programmer fills out the CR for the problem
- the responsible programmer submits the CR
  - Option: the responsible programmer puts the CR in the System Lab bin
  - Option: person working on Recent Change and Verify puts the CR in the box in the Western Recent Change Group area
  - Option: the responsible programmer gives the CR to the Field Support Group FR Coordinator
  - Option: Dept. 5426 programmer gives the CR to the Dept. 5426 FR Coordinator
  - Option: person working on Recent Change and Verify gives the CR to the Data Administration Group FR Coordinator
- the responsible programmer carries over the CR
- the responsible programmer informs the FR Coordinator that the CR has been filed

We can now be somewhat more specific about the size and difficulty of the documentation problem briefly discussed earlier. Conservatively, the information in the complete BPU for the first example above could be written in about 3 to 4 pages of English text. For 50 basic process units, this would be about 150 to 200 pages of (extremely dense) English text, to completely document FR/CR Handling in the Laboratory.

While we would then have a document, it would be almost unusable as a source of specific, directly useful information for developers. The difficulties of

looking up the needed information, and figuring out how this listing of *all* of the possible things to do would apply in the developer's particular circumstances, would be almost insurmountable. (Interestingly, we did try this approach briefly. The complete set of BPUs were given to a few highly expert users. There assessment was that the set of descriptions was far too unwieldy for practical use.)

Writing Basic Process Unit Descriptions

One of the most significant and valuable results of the MENTOR project has been the development of experience and expertise in actually writing process descriptions. The basic process unit specifies what must be given to describe a process. It is, however, totally content-free; it does not say anything about standards for what constitutes a good description.

We have developed a sizable body of heuristics, rules-of-thumb, and guidelines that address this issue. For example, one of the first rules developed was that any Name, or Stage-Option Name, should only state something that can straightforwardly occur. We recognized the need for this rule after examining an early candidate for a Stage-Option Name: ''The Laboratory finds out about a problem in a Generic''. This phrasing, quite common and normally taken to be communicative, does not state something that can actually, literally, happen. The Laboratory is not eligible to discover something; discovering is a human action, engaged in by a person. This name was replaced by ''Responsible people in The Laboratory find out about a problem in a Generic''.

The problems one encounters with poor descriptions may be divided into two categories. The first comprises descriptions that are uninformative or, worse, actually confusing, which results in developers getting answers that are uninformative or confusing. For example, it is uninformative to use purely nominal names such as ''processes the CR''. While sometimes this is the best way to say it (recall that these are rules of thumb, not hard and fast rules), someone reading this Stage Name can tell only that something is done to the CR, but not what.

The second category comprises descriptions that, by themselves, do not seem to be problems, but which are very difficult to compose and decompose (Ossorio, 1971/1978b, p. 41). For example, at one point in the pilot project we were considering having one description for all of the ways of filling out a CR—normal problem-handling CRs, Edit-CRs (which are used for an entirely different purpose and just happen to use the same form), and PIDENT-change CRs (which are used for yet a third purpose). While this certainly could have been done, the result would have been a very large and complex single description, with many Stage-Options and many Contingencies to keep things straight.

Using very complex basic process units tends to result in serious difficulty in the composition of basic process units. Recall that it is fundamental to the concept of the basic process unit that the Stage-Options Names are the Names of

other processes. If we represent what we can recognize as distinct processes in a single complex basic process unit, there is no way, within the BPU format, to refer to one of the distinct processes; it would be necessary to make some addition to the BPU. For example, a reference to a particular Version of another BPU. While this is certainly not difficult technically, it seemed to us that by far the most prudent course to follow was not to tamper with the BPU for purely technical convenience. One of the most important reasons for this choice was that the BPU is not an ad hoc invention; it is a codification of the concept of a process (Ossorio, 1971/1978b, pp. 15, 38–39). Therefore, we thought it better to write separate BPUs for distinct processes. In the case of the many ways to fill out a CR, for example, we described each of the distinct activities (''responding to a problem,'' ''giving special handling instructions,'' and ''officially changing the program considered to be at fault''), with a BPU for the process of filling out the form at the appropriate point in each activity.

An extensive discussion of these guidelines is beyond the scope of this paper. They are discussed in detail, along with the issue of how to train people to use the BPU format for description, in a forthcoming paper.

User Questions

As work on the description of FR/CR handling proceeded, it quickly became apparent that making the information available to developers would require a program to accept a person's questions and use the descriptions to answer the questions. As we noted above, the documentation dilemma is still present, whether one uses discursive English or BPU descriptions; the complete description of the software development process is so large and complex as to be almost unusable by someone needing information to act.

If we were going to write a program to answer questions, it was crucial to find out what questions developers actually had. We believed that developers' questions would fall into a fairly small number of question types, and that we would then be able to write a separate question-answering routine for each type, rather than have to address the far more difficult problem of writing a general question-answering program.

In order to find out what was actually the case, we interviewed ten developers from various parts of the Laboratory and then analyzed the results of the interviews. Each of the developers interviewed was asked what questions they had, or had had, about how the development process works. Each developer furnished about ten to fifteen questions. Half of the questions were analyzed, looking for a small number of patterns that would include all of this half of the questions. We found that eight patterns sufficed. These eight pattern questions were then checked against the other half of the questions. When the check was done, no new question types were found. As a further check, we showed the interviewees the eight pattern questions, and asked them how well they thought the pattern ques-

tions covered the range of questions they would like to be able to ask. All of the interviewees agreed that the list of questions covered the range quite well.

The analysis procedure was straightforward. We examined the lists of questions from the developers, looking for different types of questions. Frequently a question type was immediately recognizable, for example, "How do I do (something)?" or "Who does (something)?" In other cases we examined possible answers in order to get a better understanding of what the developer wanted to know, and then noticed that the question could appropriately be treated as one of the question patterns we already had. This was the case, for example, with most of the "Why did this happen?" questions from the developers, which are treated as, "How did this happen?" (This is certainly not the sort of procedure one would follow to develop a comprehensive, statistically valid model of the information needs of a population. That, however, was not what we were doing here. We were looking for adequate assurance that a program that could answer a fairly small number of questions would be suitable, if it had an adequate description of the organization. Since we knew we would be writing the program and having developers evaluate it, we considered the procedure of analysis, combined with checking our results with the interviewees, to provide adequate precautions. (See Ossario, [1981] for a discussion of the precaution-assurance paradigm of research.)

The following list of questions was developed:

1. How do I do X?
2. How do we (in general) do X?
3. What is X?
4. Who does X?
5. X has happened. What do I do now?
6. How did this X happen?
7. Who uses X?
8. What happens when I do X?

The MENTOR Program

There are two key pieces in the MENTOR project. The first is a data structure to contain the information needed. We have discussed this in some detail, and shown how the basic process unit does this, in the foregoing section. The second, equally important, is the program to access the data and use it to answer a developer's question. That program is called MENTOR. Making it possible for a person to get useful answers to questions is, after all, the purpose of documenting the software development process. The database of BPU descriptions gives all of the possibilities for doing things with whatever it describes—in this case, FR/CR handling. But a person with a question does not ordinarily want merely a list of possibilities. No one wants to go to an expert, ask a question, and be told,

"Here's all the information about that. You figure it out." To have given just the BPU database to people would have been doing this.

A good deal of clerical work is necessary to use the BPUs to figure out an answer that fits both the person asking the question and the circumstances. For example, if a person asks how to do something, one must check whether the questioner is eligible to carry out each Stage of a Version, whether all necessary prerequisite constraints have been fulfilled (which requires looking up the BPUs of other processes), which Attributional Constraints are involved, and which are satisfied. The MENTOR program does this bookkeeping and place-holding work.

A person asks MENTOR a question by typing it in ordinary English. MENTOR figures out the answer that fits the person and the circumstances, asking for more information when necessary, and answers the question in English. At the time of the pilot study, MENTOR could answer the first three of the questions listed above.

MENTOR is not, of course, actually competent in English; it cannot actually understand arbitrary English from a user. However, it will properly recognize any of the developers' actual ways of talking about the development practices. MENTOR's answers are in actual, grammatically correct English, which it composes after it has found the data to answer a question. The result is that, to a developer, MENTOR appears to understand English.

Although the BPU database forms a hierarchical description of the activities being described, the use and operation of the program is not hierarchical. A user can directly ask any question, whether high- or low-level, without having to proceed "top-down". To answer the question, MENTOR accesses only those BPUs it needs; it also does no hierarchical processing. Therefore, response time is relatively insensitive to the number of BPUs on file. (There are obvious exceptions, of course. As we shall see, to answer some questions MENTOR must check the Elements and Individuals of each BPU. This of course takes more time as the number of BPUs increases.)

Design considerations. Using MENTOR is intended to compete with the existing social practice of finding a human expert and asking. Ordinarily, when an expert is asked a question, he or she may ask for some information, and then gives the questioner an answer that is tailored to that person and the specific circumstances involved. The expert does not tell the person to do something the person is ineligible to do (e.g., sign their own CR form if they are not authorized to do so), or something that does not fit the case. Further, the expert does not ordinarily give a general format and tell the person to figure out the answer.

Rather than attempt to change the developers' habits and expectations about asking questions, it was decided at the outset to have MENTOR behave like a human expert as far as possible. Specifically, a person should be able to ask MENTOR a question with no more specialized vocabulary than one would need to talk to a human expert, and be given an answer specifically tailored to the

person and the facts of the particular case. Further, the answer is one that the person can act on, with no interpretation, figuring out, or other investigation at all. (Of course, again as with a human expert, the questioner may need more detail; people often ask an expert to give the details of how to do something. That capability is provided.)

For example, a person who asks how to fix a bug in a Generic is told:

1. Write the code to fix the problem
2. PERM the overwrite
3. Fill out the CR
4. Submit the CR

In response to a request for the details of how to fill out the CR form, MENTOR supplies the user with 13 (or 14, depending on the circumstances) steps that, when carried out, result in the CR form being filled out exactly correctly for this case.

This design differs significantly from the more common approach of writing a manual, even an on-line manual. An on-line manual will typically give a user a page or two of information, from which a user can, presumably, figure out an answer. MENTOR does not function in this manner. The reason for this choice is that this is overwhelmingly what people indicated they wanted. They do not (they reported) want to have to figure out something about some procedure that is basically (in their view) extraneous to their job. They want the immediately useful information necessary to get their job done. Many were quite emphatic about this point.

The trials seemed to bear out this decision. Most users liked being given detailed, pointed, step-by-step information. There are, of course, individual differences; of the twenty-five developers who evaluated MENTOR, three said they would prefer an on-line manual. These three were all developers who had been in the organization for six years or more.

Using MENTOR. A user wanting to ask a question calls MENTOR by typing the word "MENTOR" on the computer terminal. MENTOR responds by asking for the person's name. (It uses this to tailor the response to the individual user.)

After receiving the name, MENTOR prints the question menu. Straightforward algorithms for all of the questions have been developed. At the time of the trials three of the algorithms had been implemented. (As of this writing all have been implemented.) The menu for the trials was:

1. How do I do X?
2. How do we (at Indian Hill) do X?
3. What is X?
4. Who is X?

(''What is X?'' and ''Who is X?'' are treated identically.)

The user types the number of the question desired, and MENTOR responds with the question stem. The user fills in the stem, asking the question just as he or she would of an actual person—in ordinary English. For example, if the user types ''1,'' MENTOR immediately types, ''How do I,'' and stops, at that point on the line. The user then continues the line where MENTOR left off, typing, for example, ''fill out the CR for this problem?''

MENTOR recognizes a set of standard phrases, which are the phrases that make up the Stage-Option Names, and the names of the Elements and Individuals. In addition, MENTOR keeps a file of alternate phrasings for each of its standard phrases. If the user's input is not recognized as a standard phrase, MENTOR examines its alternate phrase lists. When it finds the user's input in an alternate phrase list, it has then recognized what the user is asking about. (In the case of an ambiguous phrase, MENTOR will ask the user which of the possibilities is meant.). For example, ''The responsible programmer fixes a problem in a Generic'' is the Name of a BPU. The standard phrases in MENTOR's files are ''the responsible programmer'' and ''fixes a problem in a Generic.'' A user might also want to say, for example, ''How do I fix a bug?'' When MENTOR encounters the phrase ''fix a bug,'' it looks it up and discovers that this means the same thing as ''fix a problem in a Generic.''

Since MENTOR understands all of the ways of actually talking about the subject matter, it appears to the user to understand English.

This very simple algorithm has proven to be adequate, because there is a relatively limited number of ways to saying the same thing in a technical organization. It is clear that MENTOR's processing of user input could be a great deal more sophisticated, including using the Classification Space technique (Ossorio, 1965, 1966), so that MENTOR could understand virtually anything a user would type in. We chose not to add this sophistication, since as long as the simple approach served there seemed little point in doing so.

After recognizing what the user has asked about, MENTOR asks for any additional information it needs in order to figure out the answer, and then gives the user the answer that fits the user and the facts at hand.

How do I do X? MENTOR looks up the BPU of the process the user has asked about. It sequentially searches the Versions of the process for a Version that this user can actually carry out in these circumstances. (This is where it uses the user's name.) To do this, MENTOR checks whether the user is eligible to carry out the Stage-Options making up the Version, and whether the Attributional Constraints on each Stage-Option of the Version are satisfied. Where necessary, MENTOR queries the user about what he or she can do and the facts of the case.

MENTOR also performs another very important check: it checks for prerequisites of this process, and queries the user about whether they have been fulfilled.

Any that have not, are flagged for the user, as a reminder, before the answer to the question is given. (This is again in keeping with the practice of a human expert answering a question; he or she knows what should have been done before this, and tells the questioner to be sure to do it first.) For example, before one submits a CR, the FR must have been filed, and a procedure called "PERM" must have been carried out. MENTOR asks about these steps, and reminds the user to first file the FR and do the PERM procedure before submitting the CR.

After answering the question, MENTOR asks the user whether he or she wants more detail. The user may ask the following detail questions:

1. How do I do the step?
2. Who does the step?
3. What is X?
4. Who is X?

Asking about details is recursive—the user may ask for sub-details, sub-sub-details, and so on. When the user is finished with detail questions at one level, MENTOR returns to questions about the previous level. (An entire line of questioning may be abandoned by hitting the "break" key on the terminal; this returns to the original question.)

How do we do X? Answering this question is less complex. MENTOR looks up the BPU for the process, prints the people who do it, and gives the user a paradigm Version of the process. No eligibility checking is done, as the user is not asking how he or she can actually do it. However, the Contingencies are checked, so that the Version fits the actual facts, as are the prerequisites.

What is X? This question is answered in two steps. First, MENTOR prints out a dictionary-style answer—an ordinary English explanation from a file. Then, if the user wants to know more, MENTOR finds all of the BPUs in which X is an Element or an Individual. For each process, it prints the Stage-Option Name in which X appears, and the Name of the process in which that Stage-Option appears. MENTOR places X in its widest context first, by printing the list of uses in order of highest level (most general) first.

Who is X? This question is treated precisely like "What is X?" A dictionary-style explanation is given, and then a list of all of the processes in which X is an Element or Individual.

At the time of the pilot study, MENTOR could answer the first three of the questions listed above. The program was approximately 3,000 lines of code in the C language. (At the time of this writing MENTOR can answer all of the questions, and has some additional capabilities as well, which are discussed

briefly in the final section. The program is now approximately 7,000 lines of C code.) MENTOR runs under UNIX (Note 1) a widely available operating system on minicomputers. On a Digital Equipment Corporation 11/70 (a large minicomputer), with approximately 20 to 25 timesharing users, typical response time is about four to five seconds.

Trials

We now knew that the descriptive technique was technically sound, we had covered complex actual subject matter, down to a technically useful level of detail, and we had a program to answer questions that users had told us were of interest to them, giving answers as a human expert does. The key question, however, remained: Would people actually use the resulting system to answer their questions?

Participants. A group of 25 users in the Laboratory used MENTOR and evaluated its answers. Of this group,

6 were very highly experienced (9 or more years),
6 were relatively new (less than 1 year),
13 were in the "mid-range" of experience,
6 were Associate Technical members of the Lab (the level below full Member of Technical Staff),
2 were in management, and
3 were specially chosen experts in some area of the Development process.

This mixture of people insured covering a broad spectrum of people throughout the Lab.

Half of the users were in the original pool of people who helped develop the user question list, and half were people with no prior exposure to MENTOR. Of the six very highly experienced developers, three had no prior experience with MENTOR, as was the case for the three specially chosen experts.

The people were given access to MENTOR, told briefly what it was for (answering questions about anything involving FR/CR handling), and invited to use it as much as they wanted to or felt they needed to to answer the evaluation questions. They used MENTOR by themselves, with no coaching other than for a brief check-back after about fifteen minutes to make sure that no misunderstanding had occurred. Average use was about one hour, with a range of one-half to over four hours.

The users then filled out a questionnaire which asked them to evaluate the quality of MENTOR's answers and, assuming that all of the user question list was implemented and that MENTOR covered all of the software development process, whether this looked like a tool they would use and recommend to others. Answers to each question were on a scale of zero to four.

The actual questions asked, and the anchoring of the scale in each case, were *as follows:*

1. How informative were MENTOR's answers? (not at all; a little; fairly; quite; very)
2. How accurate were MENTOR's answers? (very inaccurate; somewhat inaccurate; fairly accurate; quite accurate; very accurate)
3. If you were new to the Laboratory, how useful would MENTOR be to you? (not at all; a little; fairly; quite; very)
4. If you were an experienced person getting into an unfamiliar area in the Laboratory, how useful would MENTOR be to you? (not at all; a little; quite; very)
5. How much would you use MENTOR to answer your own questions? (not at all; a little; a fair amount; quite a bit; a great deal)
6. If you were mentoring a new person and MENTOR were available, how much would you use it to help the new person get on board? (not at all; a little; some; quite a bit; a great deal)
7. Would you recommend MENTOR to other people? (strongly advise against; advise against; no opinion; recommend; recommend strongly)
8. Do you agree or disagree with expanding MENTOR's knowledge to cover the whole software development process? (strongly disagree; disagree; don't care; agree; strongly agree)

Results were as follows: The mean on Question 1, on how informative the answers were, was 2.4 out of 4, or between "fairly informative" and "Quite informative."

The mean on Question 2, accuracy, was 2.7 out of 4, or between "fairly accurate" and "quite accurate." Follow-up interviews with the users about this question revealed that MENTOR actually made very few inaccurate statements (three errors were found), but users tended to rate the accuracy lower when they felt the answer should have been more complete.

The mean on Question 3, usefulness to new people, was 3.1 out of 4, or "quite useful." It is significant that there was no difference in the mean for all users, the mean for the six highly experienced users, and the new people.

The mean on Question 4, usefulness to experienced people changing areas, was 2.6 out of 4, or between "fairly useful" and "quite useful."

The mean on Question 5, how much they would use it for their own questions, was 1.9 out of 4, or approximately "a fair amount." The mean for the highly

experienced developers, who would not be expected to need MENTOR, was 1.3, or between "a little" and "a fair amount." Excluding these six, the mean on this question was 2.2.

The mean on Question 6, how much would they use MENTOR to mentor a new person, was 2.9, or approximately "quite a bit."

The mean on Question 7, recommending MENTOR to others, was 3.2 out of 4, or a bit over "recommend." Of the 25 users, 22 said they would recommend or recommend strongly; nine said they would recommend it strongly.

The mean on Question 8, expanding MENTOR's coverage, was 3.3 out of 4, or between "agree" and "strongly agree." Of the 25 users, 21 said they agree or strongly agree; 13 said they strongly agree with the expansion.

(A clerical oversight resulted in the anchoring of the scales for some of the questions being somewhat uneven, in such a way as to make the evaluations appear more negative than they would have with properly anchored scales. For example, on Questions 6 and 7, there is more distance between "a little" and "a fair amount" than between "a fair amount" and "quite a bit." There is a similar problem with Questions 1 through 4. While we would not presume to try to quantify the difference that more evenly spaced scales would make, our examination, and discussion with colleagues, leads us to believe that it is appropriate to consider the ratings conservative.)

The answer to the key question of developer reaction is that they found MENTOR to be informative and accurate, and that they would use it and recommend its use to others.

SUMMARY AND CONCLUSIONS

The pilot project was undertaken to address three critical issues:

1. Is the basic process unit adequate for describing what actually happens in developing software, completely and in detail?
2. Can we build a tool that people can use to have real questions answered?
3. If such a description and tool can be built, will people actually use it?

We addressed these issues in the strongest way possible: demonstration.

1. A complex and fairly large portion of the activities of the Laboratory (FR/CR handling) was described, completely and in detail. The content was deliberately chosen to be varied in level of detail, and was known at the outset to be one about which considerable confusion existed.

2. A program, MENTOR, was written to accept a question phrased in ordinary English and answer it in ordinary English, figuring out the answer that fits both the user and the circumstances. Further, a good deal of work was done to ensure that the questions are the ones users actually want to ask.

3. When a broad cross section of people in the Laboratory evaluated MENTOR, their assessment was that MENTOR provided accurate, informative, and useful answers, and that they would use it and recommend its use to others.

By doing this, a firm foundation for the overall MENTOR project was established. Further, we have demonstrated the feasibility and practicality of the human system approach and Descriptive Psychology technology to the problems of understanding and modeling human organizations.

It should be clear that there are a number of applications for the approach and technology. The reader is invited to make his or her own list of systems, activities, organizations, and so on, that can appropriately and profitably be viewed as human systems. We will point out only two.

First, the situation the Laboratory faces, that is, a complex process and a relatively low level of experience in that process on the part of people in the Laboratory, is significantly exacerbated in cases where the software development process is being created on a schedule only slightly ahead of the project it is intended to support. Such situations are not uncommon in the software field; any large, new software effort faces it to some extent. This obviously has a heavy impact, in terms of staff effort and calendar time, on an already tight schedule.

The second application is to documenting software. Just as MENTOR documents FR/CR handling completely and in detail, and will cover the entire software development process of the Laboratory in like manner, we can use the approach and technology to document large software systems completely and in detail. (The basic process unit, and the technology for using it that we have developed, can be used to describe any action, whether done by a human, machine, or program.) Documenting a system in this way results in a tool a person can use to find out how the system works and how to use it, from high-level descriptions down to the details of how some part of the system works. In other words, we can build a tool that can act as a human expert about the system. The ubiquity of large software systems, and the extreme value of expertise in how those systems really work, points to a high potential payoff in this area.

CURRENT WORK AND FUTURE DIRECTIONS

After reviewing the results of the pilot project, a management decision was made to carry out the full MENTOR project, covering all of the software development process and with the full capabilities of the MENTOR program. Work on the project is in progress. At this time the MENTOR program can answer all of the

questions on the user question list. The BPU database now includes approximately eighty descriptions.

The next area of the development process to be covered was the use of a critical software tool, the Source Overwrite System (SOS).

The same philosophy has been followed: MENTOR is intended to act as an expert in how to use SOS. This philosophy has an interesting implication, when applied to describing the use of software. If a human expert is asked how to use a piece of software, she or he will ordinarily give the questioner the actual commands to do the job, not a command form that the person must figure out or substitute into, and so forth. MENTOR now does exactly this. When a user asks how to use SOS, MENTOR asks for the necessary information, and then gives the user the actual commands to be entered to do the job. Thus, MENTOR now has the capability of asking a person about what he or she wants to do, and then writing the software commands to do that job.

We have found it necessary to move from the general basic process unit to a particular form, the behavioral process unit. This is a technically usable form of the intentional action analysis of human behavior (Ossorio, 1969/1978). Whereas the BPU is suitable for describing any process, from ice melting to a person behaving, describing software at a level of detail to let MENTOR actually act as an expert has called for a form of description designed for specifically human action.

With this form of description, rather than having simply Stage-Option Names, one specifies certain of the parameters of intentional action: Know, Know-how, and Performance. In the behavior of a person, Know and Know-how function as constraints; someone lacking the relevant Knowledge or Competence cannot carry out the process in question. The Performance is the actual physical performance one does, to engage in the action named by the process Name. In the case of using SOS, the Performance is entering the actual command on the computer. For example, the action with the Name "The SOS user tells TSS to call SOS" has the Performance with the Name "type SOSG program-name,CR-number."

An obvious use for the "blueprint" of the software development process is as a basis for simulation. Fundamentally, we now have the basis for saying *what* is to be simulated. The next step would be developing measures, that is, numbers representing how many Individuals eligible for various Elements there are, how long the various component processes take, how often each of the Options for a Stage is selected, and so forth. Such a simulation appears to have a high potential for tools of considerable value to systems analysts.

On a larger scale, we are now in a position to begin exploring simulation of the Laboratory in its entirety, including the practices involved in supervision, management, tool building, budgeting, feature planning, and requirements definition. Issues here would include the adequacy of the descriptive technique for cases where there is more "gray area" and judgment involved, adequacy of the interviewing procedures for gathering the information in these areas, and re-

producing human judgments. (The work of Busch [1974], Jeffrey [1975], and Johannes [1977] is relevant to these issues, and addresses both the technical feasibility and practicality aspects.) Simulation of the Laboratory would directly address the needs of management.

It seems to us that the most reasonable approach to this problem is to treat the Laboratory as a Community, as Putman (1981) has articulated the concept, including core and other intrinsic practices, subcommunities, and locutions. Simulating organizations, or other communities, seems to be a feasible endeavor at this point, and one of very wide applicability and interest.

ACKNOWLEDGMENTS

Portions of the material contained in this chapter were included in two papers presented at the conference of the Society for Descriptive Psychology in Boulder, Colorado, August, 1980. The two papers were entitled, "The MENTOR Project: Building a Rule-following Model of an Organization," and "Rules for Writing Process Descriptions." Authors' addresses: H. Joel Jeffrey, Bell Laboratories 4B-307, Naperville, Illinois 60549; Anthony O. Putman, Descriptive Systems, 1019 Baldwin Avenue, Ann Arbor, Michigan 48104.

NOTE

1. UNIX is a trademark of Bell Telephone Laboratories, Inc.

REFERENCES

Busch, E. K. *The Boulder police department: A descriptive study of organizational structure-function and individual behavior*. Unpublished M. S. Thesis, College of Business, University of Colorado, 1974.

Jeffrey, H. J. *Information retrieval by conceptual content analysis*. Technical Report 75-6, Computer Science Department, Vanderbilt University, Nashville, Tennessee.

Joahnnes, J. D. Automatic thyroid diagnosis via simulation of physician judgment. Ph.D. Dissertation, Technical Report 77-4, Computer Science Department, Vanderbilt University, Nashville, Tennessee.

Ossorio, P. G. Classification space. *Multivariate Behavioral Research*, 1966, *1*, 479–524.

Ossorio, P. G. *Dissemination research* (RADC-TR-65-314). Rome Air Development Center, New York, 1965.

Ossorio, P. G. *Meaning and symbolism* (LRI Report No. 15). Whittier and Boulder: Linguistic Research Institute, 1978. (Originally published in 1969. Boulder: Linguistic Research Institue.)

Ossorio, P. G. Representation, evaluation, and research. In K. E. Davis (Ed.), *Advances in Descriptive Psychology* (Vol. 1). Greenwich, Conn.: JAI Press, 1981.

Ossorio, P. G. *State of affairs systems* (LRI Report No. 14). Whittier and Boulder: Linguistic Research Institute, 1978. (a) (Originally published in 1971 as (RADC-TR-71-102), Rome Air Development Center, New York.

Ossorio, P. G. "*What actually happens*". Columbia: University of South Carolina Press, 1978. (b) (Originally published in an earlier version in 1971 as LRI Report No. 10a. Whittier and Boulder: Linguistic Research Institute.)

Putman, A. O. Communities. In K. E. Davis (Ed.), *Advances in Descriptive Psychology* (Vol. 1). Greenwich, Conn.: JAI Press, 1981.