MAC TR-125

A MODEL-DEBUGGING SYSTEM

William S. Mark

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

A MODEL-DEBUGGING SYSTEM

by

William Scott Mark

Submitted to the Department of Electrical Engineering on January 23, 1974 in partial fulfillment of the requirements for the Degrees of Bachelor of Science and Master of Science.

ABSTRACT

This research discusses a program which aids the user of an automatic programming system (APS) in the "debugging" of his model of his problem situation. In essence, the user must make sure that he and the APS mean the same thing by the description of the problem which the APS is to solve. The problem domain considered in this thesis is that of "business games" (i.e., the management simulation games which are used as a learning tool in the study of management). A language for describing models of these games is presented. The paper then describes the program's methods of simulating and finding bugs in models written in this language. Important aspects of the program's problem-solving approach to debugging are its internal knowledge of "bugs" and of user intention within the model. This internal knowledge stresses the importance of bugs arising from the interaction of submodels within the model. Some details of the program's implementation (in the Conniver language) are discussed. The necessity of "model-debugging" in automatic programming is emphasized.

THESIS SUPERVISOR: William A. Martin
TITLE: Associate Professor of Electrical Engineering

ACKNOWLEDGEMENTS

I would like to acknowledge the key ideas and useful criticism of Prof. Martin which were so important to this thesis. I would also like to thank Mark Laventhal for providing criticism in the later phases of this research and for taking the trouble to proofread the entire document.

Contents

1 Introduction

The purpose of this research is to explore a methodology for debugging certain models of real world situations. The models considered here consist of groups of well-defined submodels. The submodels themselves are fairly structured; the interaction between submodels is not. In this paper I will discuss a program which uses the techniques of goal-programming to explore the interactive behavior of a given model. The basic idea is that a bug in the model will give rise to a "problem". The program then tries to solve this problem in an environment defined and constrained by the model. Those steps at which the program's problem-solving process encounters constraints caused by unintended interaction of submodels suggest possible locations of bugs within the model.

To a large extent, the problems of this research are "artificial intelligence" problems. That is, the research problems involve representation of knowledge in a form which is useful to the problem-solver, and representation of the problem-solving process as a computer program. The remainder of this paper will deal with one solution of these problems for a program which debugs models of management situations. This section will present a more

complete explanation of the area of model-debugging as I see it. The next section provides an overview of the whole debugging process in the context of a detailed example. Later sections develop some ideas about bugs, problem-solving, goal-programming, and the program itself.

1.1 Define "define"

1.1.1 What is a model?

Marvin Minsky describes the concept of a "model" as follows:

> If a creature can answer a question about a hypothetical experiment without actually performing it, then it has demonstrated some knowledge about the world. For his answer to the question must be an encoded description of the behavior (inside the creature) of a sub-machine or "model" responding to an encoded description of the world situation described by the question.
>
> We use the term "model" in the following sense: to an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions that interest him about A. [12]

For the purpose of this research, the term "model" will be used in a much less general and more concrete way. Specifically, the program discussed here requires that the "encoded description" be of a particular pre-defined type, that the kinds of world-objects "A" to be modelled belong to a very limited class of things, and that "B"'s questions of

interest be sharply restricted.

After this section, the term "model" will be used to refer to a user-defined collection of constructs in a model specification language (MSL) (presented in section 4.1) which describes a "real-world" management system. (1) For now, suffice it to say that a "model" is a user's description of his system of interest. That is, the user thinks that the model describes his system--actually, the model contains bugs.

1.1.2 What is debugging?

When a model's performance is not what the user expects, we say that the model has a "bug" (see section 3). The process of finding what causes the discrepancy between performance and expectation is called "debugging". It is the nature of complex processes that the cause of a discrepancy may be related to the manifestation of the discrepancy only through a seemingly intricate chain of reasoning. The purpose of this research is to write a program which knows the necessary kind of reasoning to go from the manifestation to the cause of a bug.

---

(1) Actually, a real-world business game.

In order to incorporate this reasoning process, the program must have knowledge about MSL models (see 4.1), the kinds of bugs that occur in MSL models (see 3.3), how these bugs manifest themselves (see 4.4.2), and how the causes are related to the manifestations (see 4.4.3). Of course, this is in some sense the "whole story"; before launching into it, it might be a good idea to examine our reasons for worrying about model-debugging in the first place.

1.2 The importance of model-debugging

1.2.1 Model-debugging as a universal concept

The process of gaining knowledge about the world is a process of model formation and debugging. The progress of all organized thought, especially science, has often been described in this way. More recently, work by psychologists such as Piaget and artificial intelligence researchers such as Seymour Papert has brought this model formation/debugging view to bear on the entire learning process. Certainly, no one can doubt the importance of studying so fundamental a process.

Of course, in this research, the viewpoint must be strictly limited. The following sections

will describe a process which seems only barely related to the grandiose exaltations of the previous paragraph. For one thing, the extremely close interaction between model debugging and formation will be greatly restricted to allow examination of the debugging process itself. Also, the restrictions inherent (1) in the "show a working program" approach of this research make the class of problems seem trivial when compared to the overall problem of model-debugging.

Although I could now claim that the validity of this research effort is that it provides an initial investigation into a very hairy area (the usual induction step in artificial intelligence theses), I will move in more practical directions. (Of course, I hope for the higher parallels all along.) Specifically, I consider the importance of the kind of model-debugging actually presented here for the new field of automatic programming.

1.2.2 Model-debugging in automatic programming

---

(1) These restrictions are "inherent" at this stage of our knowledge, at this stage of my knowledge, and in the exigencies of churning out a Master's thesis. Certainly, there are no inherent restrictions in the capability of computers to incorporate the process.

Automatic programming is the art of providing a computer program (an "automatic programming system" (APS)) which takes as input some user-amenable description of a task and produces as output computer programs to accomplish that task. The user's description of his task is his "model" in the sense described in 1.1.1. This is the "model" which the program described in this thesis must debug.

But why worry about model-debugging? Why not let the user specify something, let the system generate a solution program, and simply leave it to the user to respecify the problem if he doesn't like the results? There are several answers to this question, some obvious, others not so obvious. Basically, the reasons for providing sophisticated model-debugging aids revolve around considerations of efficient use of the APS, ease of use of the APS, ease of construction of the APS, and "safety" in the use of the APS.

The most obvious reason for model-debugging is that since code-generation (i.e., actually writing the solution program after the task description is in) is a rather arduous process, it is worthwhile making sure that the user and the APS agree on what the problem is before the APS actually writes programs

to solve the problem. This idea of pre-code-generation debugging is as old as compilers, and is fairly well understood. (1)

A related but not quite so obvious reason for providing model-debugging aids in an APS is to make the system easier to use. This is especially necessary in an APS like Protosystem I [9] which attempts to provide problem-solving expertise to aid the user. The point is that the APS is designed to provide problem-solving knowledge for a user who is not at all adept in computer problem-solving. To help him design a description of his task and then not to aid him in debugging that description seems like providing not much help at all: descriptions of complex problems "always" have bugs, and finding them is usually as sophisticated a task as writing the description in the first place. (2) Thus, I beleve that an APS that does not provide model-debugging aid would be difficult, if not impossible, to use.

Supposing, then, that some kind of

---

(1) The actual debugging of models may be quite different from the debugging of source code, but the reason for doing so is the same in this case.

(2) Statistics have shown that about 50% of the time in large system development is spent in debugging [2].

debugging aid is necessary, how should it be interfaced with the user and with the APS? The answer, I think, is that debugging should occur when the system's knowledge of the user's problem is still at a high level of symbolic description. That is, prior to code generation. This leaves the debugging effort in the realm of model-debugging. The reason that it is important to keep debugging at a high symbolic level is to keep the design of the APS as simple as possible. It is quite difficult to maintain the links between mistakes which occur at low levels of description (e.g., programs) and their high-level causes. Certainly the user cannot be responsible for maintaining these links. If the APS tells him that "an illegal reference was generated from location 11437", we cannot expect him to have any notion of what went wrong with his model description. In fact, the construction of an APS which could make this connection between the bug's manifestation and its cause would be extremely difficult. It seems much more reasonable to carry on debugging at a high level of symbolic description which both the user and the APS can understand in terms of the user's model.

Finally, there is a very special problem which arises in connection with the use of the APS. The user begins to develop a <u>dependency</u> on the APS and to trust

the results of the solution programs. When the system is more expert then the user (as is the case in Protosystem I), the user may even trust results which "common sense" (i.e., previous experience, educated guesses, etc.) tells him are wrong. In these circumstances, it is of paramount importance that the user be sure that the APS has a correct understanding of his model. Other than the model-debugging subsystem within the APS, there may be no source of feedback which enables the user to find out that there is anything wrong at all . (1)

The model-debugging facility has sole responsibility for helping the user to understand what is wrong with his model in terms of the model--i.e., in the only terms the user understands. An APS which does not provide a facility for interactive discussion of the model's assumptions and their ramifications is a dangerous tool indeed. Thus, the user must always have some means of observing the effects of the assumptions in the model and for making sure that the APS "knows what he means". The model-debugging subsystem of the APS provides the necessary mechanism.

Therefore, for reasons of efficiency,

---

(1) The output code and, in many cases, the assumptions underlying its generation will be incomprehensible to the average user.

usability, and safety, a model-debugging facility is a necessary part of an automatic programming system. Still, the general problem of model-debugging in automatic programming is much too large to be considered here. In the next section, I will explain the particular subdomain of automatic programming I will attack, and my reasons for choosing it.

1.3 Details, details

1.3.1 Restriction to the WOBG

The program described in this thesis is specialized to work on models chosen from the "world of business games" (WOBG). By this I mean an environment in which the concepts common to business games are the stock knowledge. There are several reasons for choosing this domain of interest: (1) the models are sufficiently structured to be formally expressible, but are not so structured that they are susceptible to mathematical analysis; (2) the interaction of submodels is the most interesting and complex aspect of the model; (3) this is one of the few domains which is both reasonable-sized and "real-world" (in the sense that there is a great deal of interest in it independent of this research); (4) it is a

natural subdomain of the "world of business" (WOB) of Protosystem I [9].

Models in various domains differ greatly in the amount of "structure" present in the description of the model. By "structure" I mean clearly defined rules of construction and constraints on elements. The methods used in this research require well-defined models. However, if the model is "too well-defined", debugging becomes uninteresting, or is more easily accomplished by mathematical tools. The WOBG seems to have just the right level of structure. Since the idea of modelling business systems is well established, there exist a variety of formalisms for expressing business models. These modelling formalisms are even more clearly defined for business games. The very idea of a game is to have a precise set of elements and rules for manipulating them. Nonetheless, understanding and debugging models of business games is by no means trivial. There is good evidence that users of even the simplest of business games have very poor and "buggy" models of what is going on [3],[6],[8]. The main reason for this is the complexity of the interaction between submodels in business games.

I am particularly interested in debugging models in which interaction of subparts is a major

factor in model complexity. Most model-worlds which have been investigated in artificial intelligence research (e.g., the "blocks world" [21]) have few complex interdependencies. Existing interaction problems tend to be downplayed in order to emphasize other aspects of the models. (For example, see Winograd's "solution" to the "findspace problem" in [21]; cf. [17].) I wish to explore the other end of the interdependency scale; i.e., highly interactive models. (1) The kind of models which the program described in this research is designed to debug are those in which the user has a good understanding of the various parts of the model, but does not understand how these parts (which I will call "submodels") interact with each other. (2)

In fact, all of the bugs which the program is designed to find arise from interaction of submodels (see section 3.3). Business games have very

---

(1) Real world situations presumably fall somewhere in between these two extremes. However, I will devote a considerable amount of space (all of section 3) to an examination of how interaction of submodels is the major complexity factor in real world situations (in particular, in large business organizations), and how these real world interdependency problems form the "semantic roots" of similar problems in the toy-world used in this research. I am hoping to motivate an interest in the "interaction bugs" which will preoccupy the remainder of the thesis.

(2) I believe that this is a large and important class of models, including models of "systems" with well-understood elements (see [3]).

precisely defined elements (see the example game in Appendix A). However, these elements interact with each other to the extent that understanding how the "whole system" (i.e., all of the interacting parts) works is a major challenge to the players. Thus, since poorly understood interaction of submodels is the major source of bugs in this domain, the WOBG forms an excellent testing ground for my program.

Business games also have the important property of being interesting in their own right. Playing and understanding business games is considered to be an important activity at many schools of management throughout the world. There is therefore little danger of being accused of designing a program which works only in an ad hoc problem domain. Furthermore, since people are used to trying to model business games for themselves, they can appreciate the efforts of a program which aids in the debugging of such models. This "real world" flavor of business games is one of their most important properties for this research.

Finally, the WOBG is a natural subdomain of the WOB of Protosystem I. This is useful, first of all, because it allows a certain inheritance of philosophy and technique from the larger project. More importantly, though, it enables the model-debugger presented here to be

seen in the context of a large automatic programming system. Since the raison d'etre of my program is use in an APS, this connection with Protosystem I is an important aspect of the research.

Therefore, the basic philosophy of model-debugging presented here will be applied to models chosen from the world of business games. In order to show that my basic ideas about debugging are indeed "working ideas", I have written a program which uses these concepts to debug actual models of business games.

1.3.2 The role of the program in the thesis

The program presented in this thesis serves several purposes: illustration of important methods, demonstration of the workability of the techniques, and discussion of design issues for model-debugging programs. Certainly, the major use of the program in the thesis is to provide examples for the debugging theory developed in the research. All the major debugging ideas are illustrated by a scenario from the working program. As for the second use of the program, a little care is necessary in explaining the "proof" value of the program in the thesis. It is often contended that working programs prove the utility of the

theories that they represent. This is true, so long as the reader is careful not to use some sort of false induction principle to infer too much from what the program actually does. As is almost always the case, the program in this thesis can actually do only a subset of what is talked about. I will always make it clear what the program can and cannot do, how the program can be extended to do more, etc. The reader should draw any general conclusions--carefully--from this information.

Using this "program-as-illustrator" philosophy of presentation, I will now launch into a detailed example of program operation on a simple model. This will hopefully give the reader a good basic idea of what the rest of the thesis has to say. The issues raised in the example and the example itself will be discussed at length in the rest of the thesis, each aspect of the problem appearing in its logical section (see table of contents).

2 Just to give you an idea...

The important thing to keep in mind about this program is that it finds the causes of bugs in much the same way that people (or Sussman's HACKER [18]) do: by trying to solve problems--and failing. In this section I will present the complete works of my program in connection with a very simple example. A lot of new notation is presented here; please don't get bogged down in it. I present it here only to avoid vagueness in showing what the program actually works with. More complete explanations of all the notation (and indeed, the entire example) appear in the appropriate sections later on. This discussion focuses on what the program means by a "bug" and on some of the procedures used to go from the manifestation to the cause of a bug. Neither the procedures nor the descriptive mechanisms used by the program are discussed in detail here. Philosophical issues about representation of knowledge in the program and goal-programming are eschewed completely. This is a quick "introduction by doing" to the methodology of the program.

Suppose the user presents the program with the following (tiny) model:

Consider the following model of sales. A sale is a probabilistic occurence which depends only on the amount of advertising done. Advertising costs $3000 per page and is good for one quarter. I buy three pages of advertising per quarter, if the money to do so is available. Sales take place during sales calls. There is one call per salesman per quarter. A customer never buys more than one unit. If a unit is sold, the company records $5000 in accounts receivable (A-R), which is not collectable for another two quarters. At any time, a salesman has a 5% chance of quitting. If a salesman quits, a new man is hired. After three months of training, this man becomes a salesman and may start making calls. Both salesmen and trainees are paid $1000 per quarter. Trainees also have a 5% chance of quitting at any time.

The user would input this model into the program with the model specification language presented in section 4.1. In these MSL terms, the model looks like:

```
(*ACTIVITY HIRING
          (*PREREQUISITES (*PRESENT (1000 CASH)))
          (*SCHEDULE ON QUIT)
          (*PRIORITY 2)
          (*OUTPUT (SOME TRAINEE))
          (*TAKES 0)
)

(*ACTIVITY ADVERTISING
          (*PREREQUISITES (*PRESENT (3000 CASH)))
          (*SCHEDULE 3)
          (*TAKES 1)
          (*PRIORITY 3)
          (*OUTPUT (1 PAGE-OF-ADVERTISING))
)

(*ACTIVITY TRAINING
          (*PREREQUISITES
                    (AND
                              (*PRESENT (1000 CASH))
                              (*PRESENT (SOME TRAINEE))
                    )
```

```
          )
          (*TAKES 3)
          (*OUTPUT (SOME SALESMAN))
)

(*ACTIVITY SALES-CALL
          (*PREREQUISITES
                    (AND
                              (*PRESENT (1000 CASH))
                              (*PRESENT (1 UNIT))
                              (*PRESENT (SOME SALESMAN))
                    )
          )
          (*TAKES 1)
)

(*ACTIVITY COLLECTION
          (*PREREQUISITES (*PRESENT (5000 A-R)))
          (*TAKES 2)
          (*OUTPUT (5000 CASH))
)

(*EVENT SALE
          (*CONDITIONS SALES-PROBABILITY)
          (*ACTIVITIES (SALES-CALL)
                    (*OUTPUT (5000 A-R))
          )
)

(*EVENT QUITTING
          (*CONDITIONS QUITTING-PROBABILITY)
          (*ACTIVITIES (SALES-CALL)
                    (*CANCEL)
                    (*REMOVE (THAT SALESMAN))
          )
          (*ACTIVITIES (TRAINING)
                    (*CANCEL)
                    (*REMOVE (THAT TRAINEE))
          )
)

(*FUNCTION SALES-PROBABILITY
          (*ARGUMENTS (PAGE-OF-ADVERTISING))
          (*RETURN ad-function))
)
```

(I will not show the exact nature of "ad-function", as it is a *TABLE construct (see 4.1)--

just a bunch of numbers that we shouldn't worry about here (see Appendix B).)

Now suppose the user gives the program the following:

```
(*SIMULATE 4 1
          ((30000 CASH)
          (50 UNIT)
          (DON SALESMAN)
          (MARK SALESMAN)
          (STEVE SALESMAN)
          (BILL SALESMAN)
          (.05 QUITTING-PROBABILITY)) )
```

or, in words, simulate the model for 4 quarters, showing a time-slice every quarter, and with the given initial values. Before considering the actions of the program, it is worthwhile to note a few things.

First, observe that the the user has given the model (50 UNIT) as an initial resource. This is a typical example of a model-testing technique: adding slack to decouple submodels. Presumably, UNIT is something created by another submodel which the user does not wish to consider at this time. The user effectively removes this "other submodel" by making sure that the submodel is never limited by the amount of UNIT available. (The PRODUCTION submodel which creates UNIT's is shown in Appendix B.))

Second, note that we are making an implicit assumption about what the user will do with the

simulation after it is presented by the program. We are assuming that he will be either satisfied or dissatisfied with the result (1) . If he is dissatisfied, he will express his expectation to the system in the form of a goal. This initiates the debugging process. At this time, let us rejoin our example, in progress.

The first action of the program is to simulate the model as the user requests. If the user's expectation is fulfilled, no further action will be taken until the user's next request for simulation. If his expectation is not met, the program will help him find the bug in the model.

The requested simulation is shown below. The representation used here (and throughout the thesis) should be seen as a graphical description of the complex of list structure which the program uses to describe simulation histories. Every part of the diagram has an analog in the Conniver [20] representation of the program (see section 4.2).

---

(1) We are also assuming that the user is a good judge of the overall performance of the system he is trying to model. This is of course not inconsistent with our basic premise that the user does not fully understand the workings of the system (and therefore has bugs in his model). Rather, we are saying that the user knows pretty well what the model should do, but is having trouble making the model do what it should.

SIMULATION-HISTORY

TIME-SLICE 0

RESOURCES:
SALESMEN: DON, STEVE,MARK, BILL
CASH: 30000
UNITS: 50

TIME-SLICE 1

RESOURCES:
SALESMEN: DON, STEVE, MARK, BILL
CASH: 17000
UNITS: 48
A-R: 10000

SCHEDULED *ACTIVITY's:
SALES-CALL (DON)
SALES-CALL (STEVE)
SALES-CALL (MARK)
SALES-CALL (BILL)
ADVERTISING
ADVERTISING
ADVERTISING
COLLECTION (TIME-LEFT = 2)
COLLECTION (TIME-LEFT = 2)

*EVENT's:
SALE (BILL)
SALE (DON)

TIME-SLICE 2

RESOURCES:
SALESMEN: DON,MARK,BILL
CASH: 5000
UNITS: 47
A-R: 15000
TRAINEE: G0001

SCHEDULED *ACTIVITY's:
SALES-CALL (DON)

SALES-CALL (MARK)
SALES-CALL (BILL)
ADVERTISING
ADVERTISING
ADVERTISING
COLLECTION (TIME-LEFT = 1)
COLLECTION (TIME-LEFT = 1)
COLLECTION (TIME-LEFT = 2)
HIRING
TRAINING (TIME-LEFT = 3)

*EVENT's:
SALE (MARK)
QUITTING (STEVE)

TIME-SLICE 3

RRESOURCES:
SALESMEN: DON, MARK, BILL
CASH: 2000
UNITS: 46
A-R: 10000
TRAINEE: G0001

SCHEDULED *ACTIVITY's:
SALES-CALL (DON)
SALES-CALL (MARK)
SALES-CALL (BILL)
ADVERTISING
ADVERTISING
ADVERTISING
COLLECTION (TIME-LEFT = 2)
COLLECTION (TIME-LEFT = 1)
TRAINING (TIME-LEFT = 2)

*EVENT's:
SALE (BILL)

TIME-SLICE 4

RESOURCES:
SALESMEN: DON, MARK, BILL
CASH: 1000
UNITS: 45
A-R: 10000

```
          TRAINEE:  G0001

SCHEDULED *ACTIVITY's:
          SALES-CALL (DON)
          SALES-CALL (MARK)
          SALES-CALL (BILL)
          ADVERTISING
          COLLECTON (TIME-LEFT = 2)
          COLLECTION (TIME-LEFT = 1)
          TRAINING (TIME-LEFT = 1)

*EVENT's:
          SALE (MARK)
```

The simulation has resulted in 5 SALE's. Suppose that the user expected 6. There is a bug in the model--but where? Note that the model runs out of CASH in the last quarter (and therefore cannot schedule all three ADVERTISING *ACTIVITY's). However, the bug is not "NOT ENOUGH CASH". Rather, this effect is symptomatic of the bug. Most of the effort of the program is to point out bugs, not their symptoms. But this requires problem-solving in the context of the simulation history. Back to the actual action of the program...

The user notes that there were only 5 SALE's rather than the expected 6. In order to try to rectify things, the user gives the system

(*GOAL (INCREASE SALE 1))

The program is now in the debugging business. It must try

to solve the problem of increasing the number of SALE's in the context of the given simulation history. The places at which it encounters dubious constraints in the simulation environment are its possible locations for bugs.

The program uses the model and the simulation history to perform the requisite problem-solving activity for each goal as it is presented. This may be thought of as asking two questions of the model and the simulation:

(1) Why didn't you do this before?

and, if there is no good reason,

(2) How could we do this?

The method of asking and receiving answers to these questions is best explained by continuation of the example.

The first goal (given by the user) is

(*GOAL (INCREASE SALE 1))

Since this goal was given by the user, the first question is not asked. However, the second question is asked. How can we increase the number of SALE's? By examining the model and using the logic of INCREASE (explained in section 4.4.1), we see that one way to increase SALE's is to increase the probability of a SALE occuring. Thus, the system generates a new goal

(*GOAL (INCREASE SALES-PROBABILITY))

Now the program asks question number one: why wasn't SALES-PROBABILITY higher in the first place? The program looks at the simulation history and notes that the SALES-PROBABILITY was at a low in time-slice 4. Why is it so low? There was not enough ADVERTISING, the program determines. This is a BAD REASON: the model was RESOURCE-LIMITED. Okay, how can we get the necessary ADVERTISING? In order to investigate this question, the program generates a new goal

(*GOAL (SCHEDULE 2 ADVERTISING 4))

which means "try to schedule 2 ADVERTISING *ACTIVITY's in time-slice 4". (The fact that we need 2 ADVERTISING *ACTIVITY's is presumably due to the exact nature of "ad-function", and will not be discussed here.) Again, the program asks why the ADVERTISING *ACTIVITY's were not scheduled in the first place. The answer is that there was not enough CASH; still RESOURCE-LIMITED, so we pursue this line with:

(*GOAL (INCREASE CASH 6000 4))

By again asking the questions and forming new goals, the program forms the following *GOAL line:

(*GOAL (INCREASE CASH 6000 4))

(*GOAL (SCHEDULE 2 COLLECTION 4))

(*GOAL (ALLOW 2 SALE 2))

(*GOAL (SCHEDULE 3 ADVERTISING 2))

("ALLOW" rather than "SCHEDULE" because SALE is an *EVENT.) Note that we are back to SCHEDULing ADVERTISING. Are we in some kind of loop? No, we are moving back in time. Furthermore, this time, when we ask why we didn't schedule three more ADVERTISING *ACTIVITY's in time-slice 2, we find that the reason is that the user told us not to (via his *SCHEDULE specification in the ADVERTISING *ACTIVITY (see page 17)). Thus, ADVERTISING is SCHEDULE-LIMITED in time-slice 2. This is a GOOD REASON, and the program terminates action on this line of thought. Nonetheless, it saves information about the terminated line. If no more "likely" bug is found, the program will tell the user that his *SCHEDULE specification for ADVERTISING is insufficient to allow the model to meet his expectations. In the meantime, however, the program explores the model for more likely bugs. The program does this by "backing up" (1) some

---

(1) This is <u>not</u> automatic backup in the PLANNER sense. The program backs up only in certain cases, and only under program control. More importantly, the effects of the "backed-over" *GOAL's are "undone" <u>only in the context of the simulation history</u>. The terminated lines must be saved for later examination by the program. This is essential for handling the *GROUP constructs discussed later in the

and trying a different line of attack.

In this case, the program checks to see if there is another way to accomplish

(*GOAL (ALLOW 2 SALE 2))

Using its usual question-asking procedure, the program finds the alternate line

(*GOAL (ALLOW 2 SALE 2))
(*GOAL (INCREASE SALES-CALL 2 2))
(*GOAL (INCREASE SALESMAN 2 2))
(*GOAL (SCHEDULE 2 TRAINING -1)) ???

(Note that CASH does not have to be INCREASEd in this line because there is already a sufficient amount to support the new INCREASEs.) The program immediately notes that it is trying to schedule in negative time, and terminates the line.

This finishes off the entire

(*GOAL (INCREASE SALES-PROBABILITY))

idea. But there is still another way for the program to try to get that extra SALE it is looking for: by trying to increase the number of SALES-CALL's. Thus,

(*GOAL (INCREASE SALE 1))

---

thesis, and for making final debugging recommendations (see section 4.4).

(*GOAL (INCREASE SALES-CALL 2 4))

(*GOAL (INCREASE SALESMAN 2 4))

(*GOAL (SCHEDULE 2 TRAINING 1))

(*GOAL (INCREASE TRAINING 2 1))

(*GOAL (INCREASE HIRING 2 1))

(The choice of time-slice 4 for INCREASing SALES-CALL was not arbitrary: the program chooses a slice where it thinks it can do the most good.) But the program cannot accomplish this last goal. Why not? The user specifically said not to hire until someone quits. The program then checks to see if HIRING did in fact occur. Yes--one time-slice later. This particular set of circumstances suggests a common timing bug in the manager's "fire-fighting" approach to problem solving--no action was taken until it was too late for it to do any good (the solution is to anticipate problems; more details about managers' bugs in section 3). Since this bug arises from so specific a group of events, the program thinks it is a rather probable bug and gets ready to suggest it first. It then checks to see if there are any other ways of INCREASing the number of SALE's. Since there are not, it is finished looking for bugs, and is now ready to suggest the bugs it knows.

As advertised, the first bug suggested to the user is:

--BAD *SCHEDULE FOR HIRING: DEPENDENT ON QUIT; HIRING TOO LATE

The user may agree that this is the bug (I think it is), or ask the program to try again. The next bug suggested is

--BAD SENSE OF PRIORITIES: HIRING AND ADVERTISING

Essentially, the program suggests that it could have gotten more ADVERTISING if HIRING did not have higher priority. If the user doesn't buy this, the program suggests that he simply blew the *SCHEDULE specification on ADVERTISING:

--BAD *SCHEDULE FOR ADVERTISING: NOT ENOUGH

If the user still doesn't like what's happening (and since the program has suggested all of the bugs it found), the program decides to see if the user might have mis-specified or completely omitted a relevant part of his model (this happens more often than you might think) It then uses its access to WOBG knowledge to suggest

--MISSING *ACTIVITY: FACTORING

(the user may factor accounts-receivable to provide instant cash) and

--MISSING *ACTIVITY: RESEARCH AND DEVELOPMENT

(the user may increase the probability of a sale by improving his product).

The program goes out of the debugging business whenever the user takes a suggestion, or, of course, when its bag of tricks is exhausted. The user can now fix his model or change his expectations and re-simulate. Eventually, this process of simulation and debugging will converge to a model that the user is confident that he and the APS both understand sufficiently.

In this section I have tried to show a complete example of what this thesis is about. I will now go into an examination of the foundations of this approach, and the techniques that allow its implementation. I begin with a philosophical discussion of bugs (yech).

3 Bugs

A bug is something that prevents something from behaving the way someone expects it to. This section particularizes the notion of "bug" to a concept which is useful for this research. As usual, the program only knows about a narrowed-down version of "bug".

We will be interested here only in "understanding-bugs"--i.e., bugs that exist only in the user's understanding of the system he wishes to model (cf. Goldstein's "semantic bugs" [5]). This immediately removes from consideration "parenthesis errors" and other "syntactic bugs" (of course, trivial syntax bugs sometimes arise from a basic misunderstanding). Thus, there will be no interest whatsoever in finding bugs due to MSL errors. In fact, no attention is given to bugs of any kind that arise from careless expression of the user's knowledge in the modelling formalism.

The kinds of bugs with which the program is concerned are those that seem to be "inherent" in the way people understand (or misunderstand) systems. The rest of this section will be devoted to an examination of bugs that occur in the modelling process

and the features of the problem domain that cause them to occur.

3.1 Bugs in models

3.1.1 What did I do wrong?

What happens when people try to model systems? They usually do some mumbling and head-scratching and come out with some sort of expression of their ideas. In this research, the "expression" is required to be rather formal, but this doesn't matter much. Next, the modeller somehow tests his model to see how it performs under various conditions (just as my system uses simulation, see section 4.2). Most of the time, the model does not perform as the modeller expects it to--"something goes wrong".

Actually, "something went wrong" at define-time: there is something in the definition of the model which is causing the unexpected behavior. I have already mentioned the hypothesis that the user has a good understanding of each submodel. (1) Thus, the part of the model definition which is in error must be a

---

(1) The notion of "submodel" will become much more precise when I discuss MSL in section 4.1.

specification of submodel interaction. The manifestation of such a bug varies widely with the particular bug involved, and tends to be a detailed matter (i.e., highly dependent on the actual representation formalism). Therefore, I will postpone (th discussion of this problem until after I have described the formalism (4.4.2), and go on to an examination of the "semantic roots" of these "interaction bugs".

3.1.2 Interaction bugs

In order to understand the idea of interaction between submodels, it is helpful to view the model as a process which defines the action of the modelled system. Thus, the models we will examine here all "do something". The model can be seen as a system which converts some sort of input resources into some predefined outputs. (This is, in fact, a very popular view of management systems.) For the model to "do" anything, its submodels must interact with each other. That is, the inputs to the entire model are actually inputs to certain submodels which convert them into intermediate quantities which are in turn inputs to other

submodels--and so on until the desired outputs are obtained.

Via this interaction, various dependencies between submodels arise. The most common is that one submodel must wait for the completion of another before it can begin action. (See section 4.4 for a detailed account of different kinds of interaction between MSL submodels.) Also, submodels often share basic resources, giving rise to conflicts between submodels.

These dependencies and conflicts between submodels provide the environment for the following basic "interaction bugs":

(1) Unexpected conflict arising from competition for shared resources

(2) Weak performance due to poor perception of time-phased occurences

(3) Special complexity problems arising from the concentration of (1) and (2) in "tight systems" bound by higher-order constraints

Although I believe that these bugs have considerable generality, I will not discuss them in the abstract. Instead, I will move immediately into the domain of management systems to provide a framework for discussion.

3.2 Interaction in management systems

The bugs catalogued in the above subsection arise from poor understanding of complexity. This "complexity" is directly inherited by the models from the modelled domain. As an introduction to the interaction complexity of organizations in the world of business (which form the basis for business games, the "modelled domain" of this thesis), I will quote in full an illustrative passage from Galbraith [4]:

> There is considerable variation in the amount of interdependence in organizations. The kinds of variation can be illustrated by considering a large research and development laboratory employing some 500 scientists who are pursuing the state-of-the-art. Thus we have a large number of elements and high task uncertainty. However, there is little need for communication. All the projects are small and not directly connected to other projects. Therefore a schedule delay or a design change does not directly affect other design groups. The only source of interdependence is that the design groups share the same pool of resources--men, facilities, ideas, and money. But once the initial resource allocations are made, the only necessary communication between design groups is to pass on new ideas (Allen, 1969). This type of interdependence has been termed as pooled (Thompson, 1966, Pp. 54-5).
>
> If the nature of the projects is changed from 250 small independent ones to two large ones, a different pattern of interdependence arises. The large projects will require sequential designs. That is, a device is first designed to determine how much power it will require. After it is complete, then the design of the power source can take place. Under these conditions, a problem encountered in the design of

the device will directly affect the group working on the power souce. The greater the number of problems, the greater the amount of comunication that must take place to jointly resolve problems.

The second example describes a situation which is more complex and requires greater amounts of information processing. The second example has all the problems that were described in the first example. There must be budget and facilities allocations made under conditions of uncertainty. There must be a flow of new ideas among the technical specialties. But, in addition, the second example requires information processing and decision making to regulate the schedule of sequential activities. This is because there is greater interdependence in the second example.

The interdependence or interrelatedness of the design groups can be increased above what is described in the second example by the degree to which "design optimization" is pursued. Optimization means that a highly efficient device is desired and any change in the design of one of the components requires redesign of some others.

> This can be illustrated by an automobile engine and body. The handling qualities of a car depend on the weight of the engine. The engine compartment can hold only a certain size of engine with its accessories. The drive shaft and differential can handle only a limited amount of torque. Changes in the weight, size, or output of the engine may necessitate changes in the body of the automobile. These interrelations and many others must be taken into account in the design of an automobile.
>
> Actually, in the case of a passenger automobile there is a good deal of flexibility with regard to body-engine match. The engine compartment is usually large, the parts of the suspension are easily changed, and the drive shaft probably has plenty of excess torque-carrying capability. Engines of a variety of shapes and sizes are frequently placed in the same body. But this need not be the case. In

> high-performance automobiles, the size of the engine compartment is frequently sharply constrained by aerodynamics considerations. There may be efforts to lighten the whole automobile by making parts of the drive system and body as light as possible; given the required strengths. In such a situation, the flexibility in the size, shape and performance of the engine placed in the body is sharply reduced or eliminated. (Glennan, 1967)

> Thus the high performance auto is a highly interrelated system while the passenger car is a flexible, loosely coupled system. The same is true of organizational subunits which must design these systems. Any change in the engine design for the high performance car must be communicated to the group designing the body so that an optimal fit is still achieved after the change. This is less true for a passenger car. Therefore, the organization designing the high performance car must be capable of handling the information flows described in examples one and two for budgets, ideas, and schedules and also those for all design-redesign decisions deriving from the interrelated design. The amount of information that must be processed increases as the amount of interdependence increases.

Each of Galbraith's examples illustrates a kind of interdependency between subunits of an organization. The first kind, pooled interdependency, gives rise to interaction bug (1) of the previous subsection. That is, when resource sharing is present, there is liable to be unexpected conflict between subunits trying to use the same resources (These are the PRIORITY bugs of the example in section 2). Galbraith next cites an example of sequential interdependency, i.e., interaction over time

as well as resources. Again, this second kind of interdependency provides an environment for the second kind of interaction bug: when subunits interact over time, the modeller is liable to mis-estimate time-effects, thus causing degraded performance (these are the SCHEDULE bugs of the example in section 2). Finally, Galbraith mentions higher-order constraint interdependency. (1) Essentially, this means that a higher-order objective, shared by a group of subunits, has forced a need for greater interdependency among the subunits of the group. What has happened is that in the new "tighter" system, the pooled and sequential interdependency has been spread to more (sometimes all) members of the interactive group. This kind of interdependency has a direct interpretation in the WOBG which will be discussed in the next subsection. The third kind of interaction bug from section 3.1.2 of course arises from the higher-order constraint environment. (There are no examples of this kind of bug in the example of section 2; higher-order constraints were deliberately kept out for the

---

(1) I think that the introduction of the "design optimization" term here is very unfortunate. The point is that the subunits have become more interactive due to the presence of a higher-order constraint. In this case, the constraint happens to be that the units must interact in order to achieve an optimal design. However, in the next subsection I will discuss other higher-order constraints which cause the same increase in interaction.

sake of simplicity. There will be examples of this kind of bug later in the thesis.)

These three types of interdependency form the semantic roots of the bugs considered by my program. In the following subsection we will examine the way these real world organizational dependencies are modelled in the world of busines games.

3.3 Bugs in WOBG models

Business games provide a laboratory for teaching managerial decision-making. Since most important management decisions involve resolving conflicts (or possible conflicts, in the case of planning) arising from subunit interdependency, the three kinds of interdependencies discussed in the previous section are emphasized in many business games. And, of course, with the three interdependencies come the three interaction bugs.

Pooled interdependency arises from a natural sharing of resources by different parts of the game-player's "business". The business game contains a very well-defined set of "resources" (cash, salesmen, production-lines, etc.) which the player must manipulate accord ng to certain specified rules of play. (1) The basic

idea is to accumulate certain resources which are designated as "assets". There are a variety of strategies for accumulating assets (e.g., use research, do some advertising, learn about market trends, etc.). The important point for us is that the implementation of any strategy requires manipulation of various subunits of the player's "business". These subunits share the pooled resource of cash. Since cash is in limited supply, an interdependency is set up, and conflicts arise. Poor understanding of this pooled interdependency gives rise to section 3.1.2's bug type (1): "unexpected conflict arising from competition for shared resources."

A much more interesting aspect of the particular game I have selected is the sequential interdependency among subunits. First of all, note that some of the activities of the subunits are "long-term" (research and development, training sales personnel, constructing additional production capacity, etc.), while others are "short-term" (advertising, factoring accounts receivable, hiring, etc.). Second, there is considerable linkage between the requirements of some activities and the

---

(1) This discussion is based on the actual business game presented in Appendix A--it might be a good idea to glance over the description of the game to give yourself the flavor of what's going on.

"outputs" of others (production provides units to sell, hiring provides employees to train, etc.). Finally, the game contains a rather rich "possibility space" for any given strategy if the time-scale is long enough. That is, there are a variety of non-independent ways of going about achieving a given task over time. All of this (plus the addition of probabilistic occurences over time) adds up to a complex pattern of sequential dependecies, which in turn gives rise to bug (2), "weak performance due to poor perception of time-phased occurences".

It is characteristic of the game used here (and of most other business games) that the pooled and sequential interdependencies are frequently made more intense by "higher-order constraints". These constraints arise from the activity structure of the game. The key factor is that various activities and functions of the organization depend on the outputs of <u>more than one</u> prior activity (note that this was not the case in the example of section 2, and thus this problem was avoided). I can present a detailed account of these mutual interdependency relationships only after I discuss the way the game is modelled in MSL (I will do this in 4.4). For now, it will suffice to say that two kinds of higher-order constraints are distinguished: the kind in which several activities (or,

more usually, chains of activities) must combine to provide resources for another activity, and the kind in which a number of activities can combine in various unstructured ways to achieve a functionally-determined goal.

This section has been devoted to filling in rather general background information about the kind of bugs the program knows about and how these arise naturally in real world systems. We now go on to an examination of how the program incorporates some knowledge about these bugs, and how it goes about using this knowledge to debug models.

4 How the program works

In this section I will present a program which finds the kind of interaction bugs discussed above. An example of program operation has already been shown in section 2. From this example, the following pattern of program operation is evident: the program starts with a model represented in a special formal language; it takes this model and produces a simulation of it; if the user finds a discrepancy between his expectations of model performance and the results of the simulation, he presents the program with the goal of eliminating the discrepancy. The program then attempts, using both the model as originally stated by the user and the results of the model's simulation, to achieve that goal; in the course of failing to achieve that goal (1) , the program finds features of the model which it considers to be unintended causes of the failure--bugs. it then suggests these bugs (in order of "likelihood") to the user, leaving him to take the next step (and perhaps re-initiate the process).

This section considers each aspect of

---

(1) The program should fail to achieve almost all user goals! (The "almost" is due to probabilistic considerations.) Otherwise, there was not a bug and the simulation would have achieved the goal in the first place.

this process in turn. It begins (4.1) with an examination of the model specification language, providing a firm basis for understanding what the program does and does not know about the user's model. Next (4.2), it describes the simulation of the model and the way the results of the simulation are presented to the debugger. Continuing along the debugging process, section 4.3 deals with the way user goals are formed and the way in which the system handles goals. Section 4.4 can then talk about how the program's deductive mechanisms pursue goals and locate bugs--the real guts of the debugging problem. Finally, there is a short section (4.5) on the way the program uses real-world knowledge in the debugging process.

Into the heart of darkness...

4.1 The model specification language

In order for the program to use the simulate-and-investigate method for debugging models, the models must be represented in a form that is executable (by the simulator) and a form that is examinable (by the problem-solving routines). The model specification language (MSL) is an attempt to combine these two necessary forms in a single language (which also purports to be fairly

user-oriented!).

MSL is a set of simple primitives which can be used to describe models--especially business game models (1) . An MSL specification consists of an (unordered) collection of the three basic primitives *ACTIVITY, *EVENT, and *FUNCTION. The basic primitives are further described by modifying constructs. The model manipulates user-defined value/term pairs called "resource variables" (e.g. (1000 CASH), (SAM SALESMAN), etc.). An example of MSL specifications appear on pages 17-18, and in Appendix B. This section contains a brief description of the syntax and semantics of these MSL primitives.

The basic MSL construct is the *ACTIVITY. The concept of "activity" used here is precisely similar to the usual business sense of the word: a well-defined organizational task which processes some commodities or information that is used by the organization (see section 3.1.2; see also the WOB [9] for its information on activities). An *ACTIVITY also corresponds to a submodel (2) --that thing that the user is supposed to have a good

---

(1) No claim is made for any "completeness" or "sufficiency" of this set of primitives. These are simply constructs which can be used to express my game models.

(2) We will see in a few minutes that *EVENT's and *FUNCTION's are also submodels.

grasp of (see 3.1.1).The *ACTIVITY specification looks like

(*ACTIVITY <*ACTIVITY-name> <modifiers>)

(1)

As is usually the case, the modifiers are the most interesting part of the specification.

One modifier which is almost always present is the *PREREQUISITES specification. This construct expresses the necessary inputs of an *ACTIVITY.

The *PREREQUISITES specification contains an arbitrary number of

(*PRESENT <resource variable>)

forms grouped (implicitly) by OR or (explicitly) by AND. The basic interpretation is that the named <resource variable> must be present (2) for the *ACTIVITY to be initiated. If there is an AND specification, then (as one would expect) all of the "AND'ed" resource variables must be *PRESENT. Thus, in

---

(1) I will use the following notation: "<" and ">" are metalinguistic brackets which surround metalinguistic statements. Everything else belongs there.

(2) Clearly, there are the obvious extensions "*MAY-BE-PRESENT", "MUST-BE-PRESENT", etc. I have not found these concepts necessary to express the models I have used. Therefore, they are not included in the MSL, even though their introduction would be straightforward.

```
(*ACTIVITY SALES-CALL
 (*PREREQUISITES
  (AND
   (*PRESENT (1000 CASH))
   (*PRESENT (1 UNIT))
   (*PRESENT (SOME SALESMAN))
  )
 .
 .
 .
          ) )
```

there must be (1000 CASH), (1 UNIT), and (SOME SALESMAN) for SALES-CALL to be initiated.

Some further comment is necessary on the quantification mechanism of *PRESENT. The "SOME" in (SOME SALESMAN) represents any name of a SALESMAN in the model.That is,

(*PRESENT (SOME SALESMAN))

will be satisfied with

(MARK SALESMAN) or

(DON SALESMAN) or

(STEVE SALESMAN)

Numerical quantifications carry an implicit "at least" modifier. That is,

(*PRESENT (1000 CASH))

will be satisfied with

(10000 CASH) or

(1000 CASH)

but not (999 CASH)

The "at least" modifier may be explicitly stated, or may be changed to AT-MOST, as in

(*PRESENT (1000 CASH) AT-LEAST)

(*PRESENT (5 ERRORS) AT-MOST)

The "outputs" of an *ACTIVITY are expressed by the *OUTPUT and *REMOVE constructs:

(*OUTPUT <resource variable>)

(*REMOVE <resource variable>)

which add or delete the named resource variable from the model's resources.

An *ACTIVITY construct may be further described by:

(*TAKES <number>)

to indicate that if the *ACTIVITY is initiated in time-slice n, its outputs do not become available until time-slice

n+<number> . The purpose of this is, of course, to allow the modelling of *ACTIVITY's which take an appreciable amount of time to be completed. Another important modifier,

(*PRIORITY <number>)

allows the user to indicate preference in allocation of resources to *ACTIVITY's. Thus, if several *ACTIVITY's are vying for the same resource, the one with the lowest *PRIORITY <number> has first crack at it (1) .

*SCHEDULE specifications allow the user to give explicit scheduling information to the simulator in order to limit the use of an *ACTIVITY. The specifications that have been found useful so far are

(*SCHEDULE <number>)

to limit the number of times an *ACTIVITY can be scheduled in any time-slice,

(*SCHEDULE (ON <*EVENT-name>))

to allow the scheduling of an *ACTIVITY only in the same

---

(1) Again, obviously, this simple mechanism could be greatly expanded. More complex models would require time-varying and other computed *PRIORITY specifications. These have not been included in MSL.

time-slice as the occurence of the named *EVENT, and

(*SCHEDULE (EVERY <number>))

to limit the scheduling of the *ACTIVITY to time-slice <number>,2x<number>,3x<number>,etc.

The above modifiers, along with the user's ability to create resource variables and provide arbitrary *ACTIVITY structures, allow enough flexibility to express all of the *ACTIVITY's necessary to model the game in Appendix A (see the model in Appendix B). There are, however, other kinds of submodels to be considered.

Another basic construct (i.e., submodel-specifier) available to the modeller is the *EVENT. This is used to express parts of the model which are "outside of the system"--beyond the organization's direct control. These outside influences are often modelled as probabilistic occurences, so that *EVENT's are usually associated with the probabilistic parts of the model. *EVENT is very similar to *ACTIVITY in basic syntax:

(*EVENT <*EVENT-name> <modifiers>)

but the modifiers are somewhat different.

Instead of the *PREREQUISITES specification, a *CONDITIONS list is stated:

(*CONDITIONS <boolean expression>)

That is, the simulator expects the body of a *CONDITIONS list to evaluate to "true" or "false". Usually, the body contains some combination (perhaps related by AND or OR) of *FUNCTION names (1) (see below). The intent is that the *EVENT may not be initiated unless the <boolean expression> evaluates to "true".

Usually *EVENT's affect particular *ACTIVITY's.The suscteptible *ACTIVITY's and the actions to be taken by the *EVENT are expressed within the *EVENT by the *ACTIVITIES modifier:

(*ACTIVITIES (<list of *ACTIVITY-names>) <actions>)

If an *EVENT contains an *ACTIVITIES construct, it can be initiated only in a time-slice in which at least one of the named *ACTIVITY's is scheduled.

One rather unusual <action> which can be taken by an *EVENT is

---

(1) These *FUNCTION's usually express a probability with which the *EVENT occurs in a given time-slice. The simulator sets up a probabilistic event (no confusion, please!) on the related sample space to express the *FUNCTION. It then calls a random number generator. If the value returned by the RNG falls within the defined event,the simulator assigns "true" to the value of that *FUNCTION.

(*CANCEL)

This means that the interrupted *ACTIVITY has been permanently disrupted, and is to be unscheduled. (Of course, it can be rescheduled later.) In all other respects, *EVENT's are treated just like *ACTIVITY's.

The final basic construct in MSL is *FUNCTION. It expresses a functional relationship between variables in the model, and, in general, accounts for information flow within the model. It is thus slightly different in spirit from the resource-handling *ACTIVITY's and *EVENT's. Nonetheless, it shares submodel status (1) , and is similar in syntax to the other two basic constructs:

(*FUNCTION <*FUNCTION-name> <modifiers>)

*FUNCTION's are not "scheduled"; rather, they are invoked by being mentioned in other constructs (just as in programming language function calls). Thus, whenever SALES-PROBABILITY (see section 2) appears in the model (except in the *FUNCTION definition, of course), the *FUNCTION

---

(1) It is important to recognize that information-handling activities are submodels at the same level as other organizational activities. Forrester stresses this point in [3], and seems to use the homogeneity of basic submodels successfully. Of course, the uniform submodel constructs also lead to a gain in modelling efficiency and a lessening of the cognitive load of the MSL user.

SALES-PROBABILITY will be invoked.

The analogous construct to *PREREQUISITES and *CONDITIONS in *FUNCTION is

(*ARGUMENTS <argument1> <argument2> ...)

which behaves like the usual argument-list in programming language functions. Missing arguments cause an "error" which stops the simulation (1) .

The analogy to *OUTPUT is

(*RETURN <expression>)

where <expression> can be a combination of *FUNCTION names and the special function-representing constructs

(*TABLE (<*ARGUMENT-name> <*RESULT-name>)
<argument/result pairs>)

(*SUM-UP (<variable range>) <linear factors>)

This is about all there is to the MSL. The semantics of *ACTIVITY's and *EVENT's are developed a bit further in the next section. *FUNCTION's are dealt with in 4.4.2.1. However, no really detailed descriptions are presented anywhere. There is little point in it. The only

---

(1) This is, of course, the kind of bug we're <u>not</u> interested in here.

purpose of presenting MSL is to allow the reader to understand the examples and judge what the program does and does not know about a particular model.

Almost all of what the program knows about any given model is in the MSL specification. (It knows a few other things discussed in 4.5.) MSL can be simple because the models considered are quite simple. As the models become more complex we expect (by conservation of complexity) that MSL will become more complex. The hope is that MSL contains something general enough to handle some kinds of additional model complexity without additional language complexity. This "something" is the basic philosophy of submodel structuring which is reflected in the MSL. Thus, I have tried to emphasize this basic structure rather the details. In the next section we follow the course of the program's debugging process and examine the simulation of MSL models.

4.2 Simulation as a way of doing things

Simulation is a technique for observing the behavior of models. In the absence of analytical and other "high-level" tools (like educated guesses), simulation is the only way to find out wha a model "does" in any given

situation. In the model-debugging system presented in this thesis, the simulator sets up the basic feedback mechanism between user and APS.

At the very least, any APS should provide a facility for checking out model behavior with simulation. That is, the user formulates his model, tests it via simulation, changes it if he doesn't like what he sees, and resimulates. For reasons discussed in the introductory section, it is necessary to go a step further. The program described here attempts to aid the user in discovering why the model does not perform as he expects it to .

Therefore, this section will concentrate on simulation as a way of initiating the debugging process. This emphasis ignores very important issues of presenting simulation results to the user. In fact, it completely downplays the importance of the simulator itself, concentrating only on the interaction of the simulator and the deductive mechanisms of the debugging program. Thus, in this section I will proceed to finesse the simulator and move on to the more relevant problems of representing the knowledge gained by the simulation in such a way that it can be used by the debugger.

4.2.1 The simulator finessed

In this section I will very briefly describe the simulation scheme used in the program. The whole simulation philosophy presented here is kind of strange as viewed from the standpoint of "normal" simulation programs. This is due to the presence of two major design criteria not usually found in the area of simulation programming:

(1) Adherence to the "user only knows local submodel information" canon ennunciated earlier (sections 1.3.1 and 3.1.1)

(2) The goal of representing knowledge found by the simulation in such a way that it can be used by the debugger

The first criterion gives rise to those funny MSL constructs which mysteriously appeared in the previous discussion. It also motivates the style of simulation described in the rest of this section. The second criterion determines the actual implementation of the algorithm, and is dealt with in the following subsection.

In MSL, the information pertaining to a

particular submodel is found only in that submodel. The kind of "information" varies from submodel to submodel (as described in 4.1), but basically, the following specifications are necessary:

--resources needed by the submodel

--resources produced by the submodel, and the length of time necessary to produce them

--explicit policy for the conditions under which the submodel should be activated

The basic operation of the simulator is then straightforward.Each submodel is activated when its (user-specified) explicit pre-conditions are satisfied, provided that all of its necessary resources are available. If the user does not specify pre-conditions (via *SCHEDULE and *CONDITIONS--see 4.1), the submodel is activated whenever its necessary resources are available (subject to *PRIORITY restrictions, of course). When the time (specified by *TAKES) for submodel activity has elapsed, the output resources of the submodel (if any) become available to the whole model. This process of cycling through submodels activating "ready" ones, continuing "running"

ones, cleaning up finished ones, and augmenting and depleting resources all along continues for the duration of the user-specified run-length.

Now anyone who has ever glanced at the guts of a simulator knows that I have just finessed inumerable details (as well as a few important Points). The algorithm used in the program is actually a bit more sophisticated and a great deal hairier than the one "described" above. For example, I have not even mentioned the rather ticklish problem of handling probabilistic occurences in this context, nor the design decisions for priority-scheduling of already-running submodels. I am deliberately stuffing the details here because the simulator itself is not very important to the thesis as a whole. It is its output, the SIMULATION-HISTORY context, that I wish to emphasize here.

4.2.2 Simulation history and SIMULATION-HISTORY

The form of the output of a simulation program is always a key factor in its usefulness. In the debugging system presented here, it is an essential link between the model and the deductive mechanisms of the

debugger. As discussed above, much of the task of the simulator is to present the knowledge gained by simulating the model in a form that can be used by the rest of the program. This is of course the old artificial intelligence task of representing knowledge in a form that can be used by procedural deductive mechanisms.

The style of representation I have chosen for the simulation knowledge is the simulation history. Now this is hardly startling--simulation histories are frequently used to describe the behavior of systems. But here I wish to extend the concept somewhat. In my program, the simulator constructs a simulation history (called SIMULATION-HISTORY) which then becomes the problem-solving environment of the debugger. By this I mean that from the point of view of the deductive mechanisms in the debugger, the "world" is a simulation history; i.e., a sequence of facts about the model which are true at various times determined by the simulation. The debugger lives inside this simulation history. The things that it knows about the "world"--the kinds of knowledge found, the way events are related, etc.-- are the facts and rules of the simulation history world (1) . In thinking about the

---

(1) Except for, as we shall see later, the facts it knows about the "real world" of business games.

debugger, it is well to keep in mind that it is a citizen of the simulation history world.

Well then, let's go slumming and look around the simulation history world ourselves for a few rollicking moments. Consider some set of observational variables on a simulation model. Then a simulation history can be thought of as a recording of the "values" of these variables at various instants of simulation-time. The interesting questions are what observational variables should be used and how the record should be organized. We will see that these questions are important with respect to the usefulness of the simulator to the debugger.

For the simulation to progress from one time instant to the next, the simulator must have a record of the state of the simulation. The simulation state of these simple MSL models consists of four main pieces of information:

(1) the value of each "resource variable" (see 4.1) at the end of each time-slice (1)

(2) a record of the *ACTIVITY's which were initiated in the time-slice

---

(1) A time-slice is one ker-chunk of the simulator.

(3) a record of the *EVENT's which occur and the *ACTIVITY's they affect

(4) an indication of the stage of completion of each "running" (i.e., previously initiated and not yet complete) *ACTIVITY and *EVENT

Therefore, the simulator needs these four pieces of information at the end of each time-slice in order to go on to the next time-slice.

But what does this have to do with the "observational variables" for the simulation history? First, remember that the "observer" in this case is the deductive mechanism of the debugger. Now, harking back to all that was said in sections 1 and 2 about debugging by problem-solving, we can see that the debugger is usually in the position of trying to change the course of the simulation in some way (to cause some desired outcome which causes another desired outcome, etc... which eventually causes the user's desired outcome). In order to decide whether it can make the change (1) it must know something

---

(1) Of course, it must also decide whether the user wants the change to be made. This part of the problem is discussed in 4.4.2.

about the simulation. Specifically, it must know the state of the simulation and ways to change that state (1) . The ways to change the state are encoded in procedural deductive mechanisms to be described later (4.4.1). The state of the simulation can be provided by the simulation history. Therefore, the observational variables for the simulation history are just the state variables discussed above (2) .

Well, since the simulator needs the values of the state variables at the end of each time-slice, the program need only keep track of these values in some useful fashion. The problem now becomes one of organizing the simulation history. In order ot think about such an organization, we can look back to section 2 and remember a bit more about what the deductive mechanisms do with the simulation history.

The deductive mechanisms usually find themselves playing around in their little simulation history world in two ways:

(1) examining a single time-slice to see whether a change can be made at that time

---

(1) This is its "world knowledge" of the simulation history world.

(2) A schematic representation of these state variables as they appear in the simulation history is found on pp. 21-23.

(2) examining a large segment of the simulation to choose a likely time-slice for scheduling something new, to follow the course of an *ACTIVITY or *EVENT, to pursue the consequences of a proposed change, or (as we shall see later in this section) to handle higher-order constraints

What we need is a good representation for facile handling of time-slices and (usually contiguous) groups of time-slices. The representation should also allow ease in the building-up and manipulation of the whole history.

Such a representation is the Conniver context [20]. The simulation history is implemented as a Conniver context with the unlikely moniker of SIMULATION-HISTORY. Each time-slice is a layer [20] of the context. This Conniver implementation implies the following relation between time-slices: the simulator "grows" SIMULATION-HISTORY by adding on new time-slices; changes made to the data in a new time-slice are invisible to earlier time-slices, however, the status of any datum can be determined in any time-slice. This certainly gives us the record of the simulation history that we want. Conniver also allows any part of the context to be regarded as a separate context. The importance of this is that the context can then be used as the database, or, more

precisely, as the working environment, for some set of programs. That is, the programs in a given context work only with that context as a knowledge base. Thus, we can see that the deductive mechanisms of the debugger can "live inside" the simulation history by simply using SIMULATION-HISTORY as their context. Furthermore, the deductive mechanisms can live inside any part of the simulation history which they must examine. Their world can be a single time-slice or a large, program-edited piece of the history.

We will see that this ability to live inside arbitrary pieces of SIMULATION-HISTORY is a key requisite for the deductive mechanisms of the debugger. For the deductive mechanisms to work, they must apply their procedurally-embedded knowledge of how to change the course of the simulation to carefully chosen parts of the simulation. This is why the simulation history and its implementation as SIMULATION-HISTORY form such an important part of the program. In the next section, we will find that the SIMULATION-HISTORY representation gains further importance when the debugger generates hypothetical states of the simulation.

4.3 Goals and environments

Throughout the thesis I have been using the word "goal" to describe a variety of phenomena. I have spoken of user goals, system goals, and submodel goals. In section 2 I introduced another construct containing the word "goal":

(*GOAL <strange words> <numbers> <lots of parentheses>)

which purported to represent the various other kinds of goals to the program. In this section I will discuss what these parenthetical thingees mean to the program. In the next section I will talk about how they are created and manipulated. Here I describe only goals qua *GOAL's--i.e., the common structural aspects of *GOAL's.

A goal expresses a desired state. In a debugging context this desired state is almost always inconsistent with the actual state. This is because the user has found a discrepancy between reality and expectation and has thought of a desired state in which the discrepancy is resolved. Thus, the desired state, reflecting the fixed discrepancy, is inconsistent with the actual state. In the program presented here, the user can ask the program to produce this desired state (given the model and the simulation history--see section 2). (1) The request is made

---

(1) As discussed elsewhere, the program fails in its attempt

via a *GOAL statement:

(*GOAL <achieve desired state>)

What does it mean to "achieve the desired state"? The user is asking the program to change the course of the simulation. The program goes about this by first creating a hypothetical simulation state (time-slice) which includes the desired state. Then it attempts to make the rest of the simulation history (i.e., the previous time-slices) consistent with the new hypothetical time-slice. (1) This is done by the creation of a new *GOAL

(*GOAL <make previous time-slice consistent with new one>)

This new *GOAL is clearly of the form

(*GOAL <achieve desired state>)

and can thus be handled exactly like the user goal. The program can thus recurse merrily along until it cannot achieve a desired state--i.e., until it fails.

Now then, let's take a closer look at

---

to produce the desired state, but this is not important to the discussion of this section.

(1) This "work backwards" methodology is due to the debugging philosophy of tracing a bug from its manifestation back to its cause.

this process. Each *GOAL requests a specific change to a specific local environment (the time-slice). Thus, each *GOAL is attempted in the context of a local constraint environment represented by a single time-slice of the simulation history. (1) If the *GOAL is achieved, it will define a new environment which is inconsistent with the old time-slice (because of the changes wrought by achieving the *GOAL). This new environment is then consistent with the user's desired state, but inconsistent with the old simulation history. The program will then use this new local environment as a basis for defining the next desired state along the line toward making the whole simulation history consistent with the user's desired state. The program is, in effect, constructing a new hypothetical simulation history which results in the user's desired state. (2)

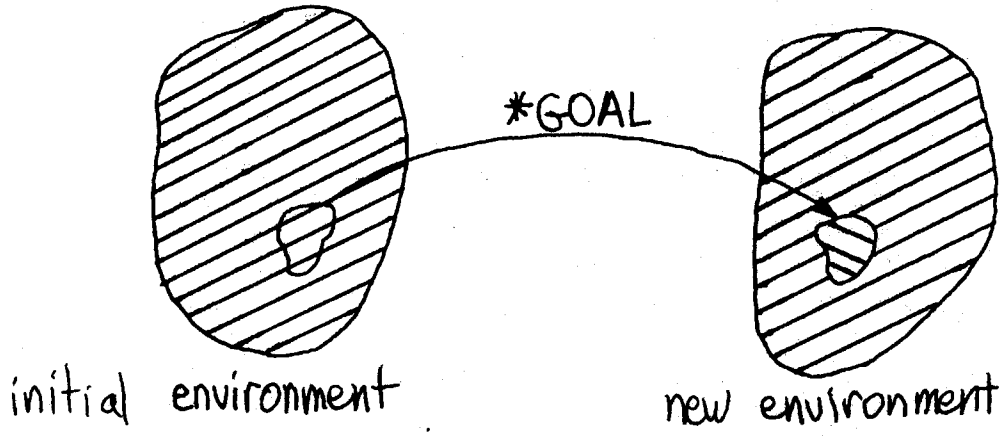
Thus, environments are intimately related to the semantics of *GOAL's. Each *GOAL is constrained by a pre-specified part of the simulation

---

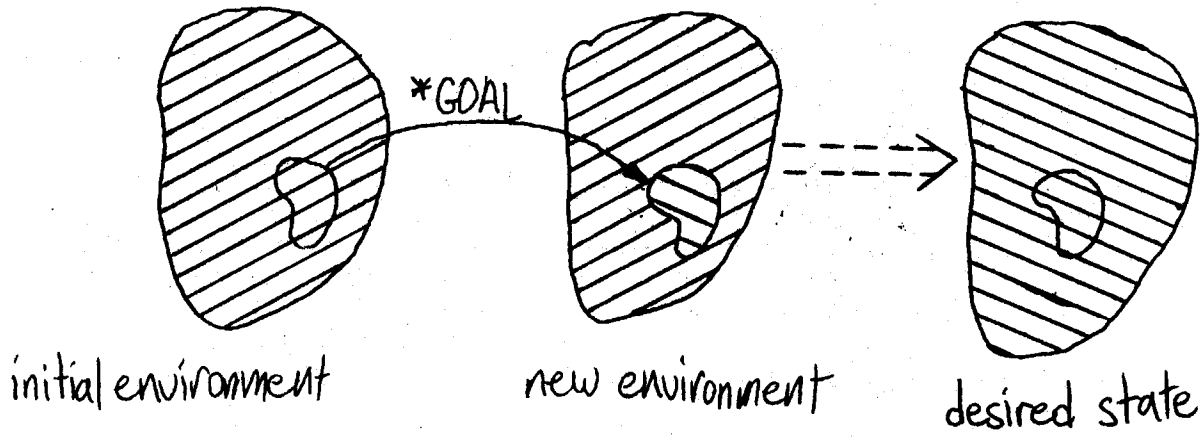
(1) Not quite. As we shall see in a second, multiple goals are achieved with respect to a local constraint environment consisting of several time-slices.

(2) The next section deals with the problem of how the program constructs this simulation without destroying the original intent of the model. Specifically, section 4.4.2.1 gives a better picture of what is "constraining" about a "local constraint environment".

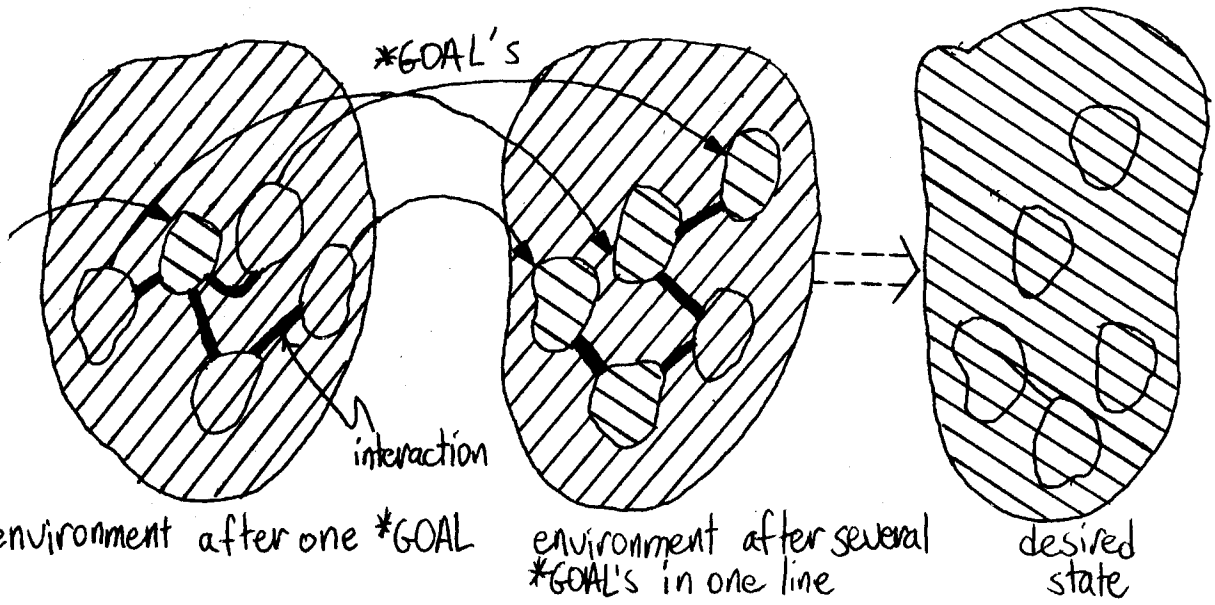
environment--that part which it is supposed to change. The achievement of a *GOAL can therefore be seen as a transformation:



This transformation is a local phenomenon. However the effects of the transformation are non-local. The *GOAL has perturbed the local environment and made it inconsistent with the global environment. Since the eventual goal of the problem solver is to create a consistent simulation history which results in the user's desired state, the global environment must be made consistent with this new inconsistent piece:



In order to make the global environment consistent, the program must trace down the effects of changing that local piece. In other words, it must examine the way that local piece interacts with other pieces of the global environment:



But this is exactly what we want. The user is incapable of following the interactions of the model. If the program is to help the user find the "interaction bugs" thus created, it must have some mechanism for tracing interactions. This mechanism is the problem-solver.

The problem-solver uses a *GOAL to express a global environment perturbation. It then uses the deductive mechanisms described in the next section to follow that perturbation throughout the local environment, the local change at each point being determined by a *GOAL.

When the program comes to a point where the perturbation cannot be continued (i.e., where a *GOAL fails), it has, in effect, discovered a part of the environment which cannot be made to conform to the user's desired environment. It has traced the interaction path to its roots--it has bracketed the bug location between the user's desired simulation state and the user's desired constraint which gave rise to the interaction (see 4.4.3).

Thus, *GOAL's are the vehicle for exploring the interactive behavior of the model. As we have seen above, the use of *GOAL's in this way requires sophisticated manipulations of local environments. In order to tie down some of the concepts discussed in the previous paragraphs, I will now discuss some of the problems the program faces with respect to this environment-handling.

First, each *GOAL must be achieved with respect to a local environment. That is, the *GOAL must only "see" the constraints of a local environment (not the whole thing) (1) , and must directly affect only that local environment. Otherwise, the distinction between local and interactive behavior is lost--there is no such thing as a

---

(1) This is due first to the nature of the problem-solving process--"set up a local environment and then make the next local environment up the line consistent with it"--and second to the debugging philosophy espoused in 4.4.2.1.

"perturbation".

Fortunately, the environment to be examined is the SIMULATION-HISTORY context. We will see in 4.4.2.1 that the required local environment is (usually) just a TIME-SLICE of the SIMULATION-HISTORY. The *GOAL can thus be made to "see" only a local environment by making the required TIME-SLICE its working environment (as in 4.2) (1). The context structure makes the relation between TIME-SLICE's evident (i.e., because each is a Conniver layer), so that the distinction between local and interactive constraints is explicit in the built-in (Conniver) semantics of SIMULATION-HISTORY.

Now the *GOAL must also be made to affect only a local environment if the semantics discussed earlier are to be preserved. It would seem that this is just as easy: simply keep the TIME-SLICE in question as the *GOAL's working environment, and all changes will explicitly have the required locality. However, there is a complicating factor found in all searching problem-solvers: the problem-solver must make provisions for discarding an old line of attack and beginning a new one. This is the old problem of backup which has been discussed extensively in

---

(1) This isn't quite so simple for multiple *GOAL's, as we'll see in a second.

|7| and |19|.

The backup problem is germane to the debugging process because the debugger usually attempts to find all possible causes of a particular discrepancy (in the hope that one of them is the actual bug). Thus, it will follow down one line of attack, fail, and try another. It must therefore be ready to erase the consequences of the line to be discarded. But this is a particularly hard problem for the debugger. Here, the tracks leading to failures are the key to the rest of the process. They cannot be simple "erased", but must be preserved in some form which the program can use to suggest bugs and to explain its actions to the user see 4.4.3).

Furthermore, while the effects of each *GOAL must be resticted to a local environmet, the effects of all the *GOAL's must create a new consistent environment (1) . Thus, the program must maintain some new environment which localizes the effects of the *GOAL's, allows a controlled backup with preservation of the backed-over information, and which forces consistency of all affected environments. Certainly, SIMULATION-HISTORY will not do.

But something like it will. The program again uses a layered-context structure. In each layer it

---

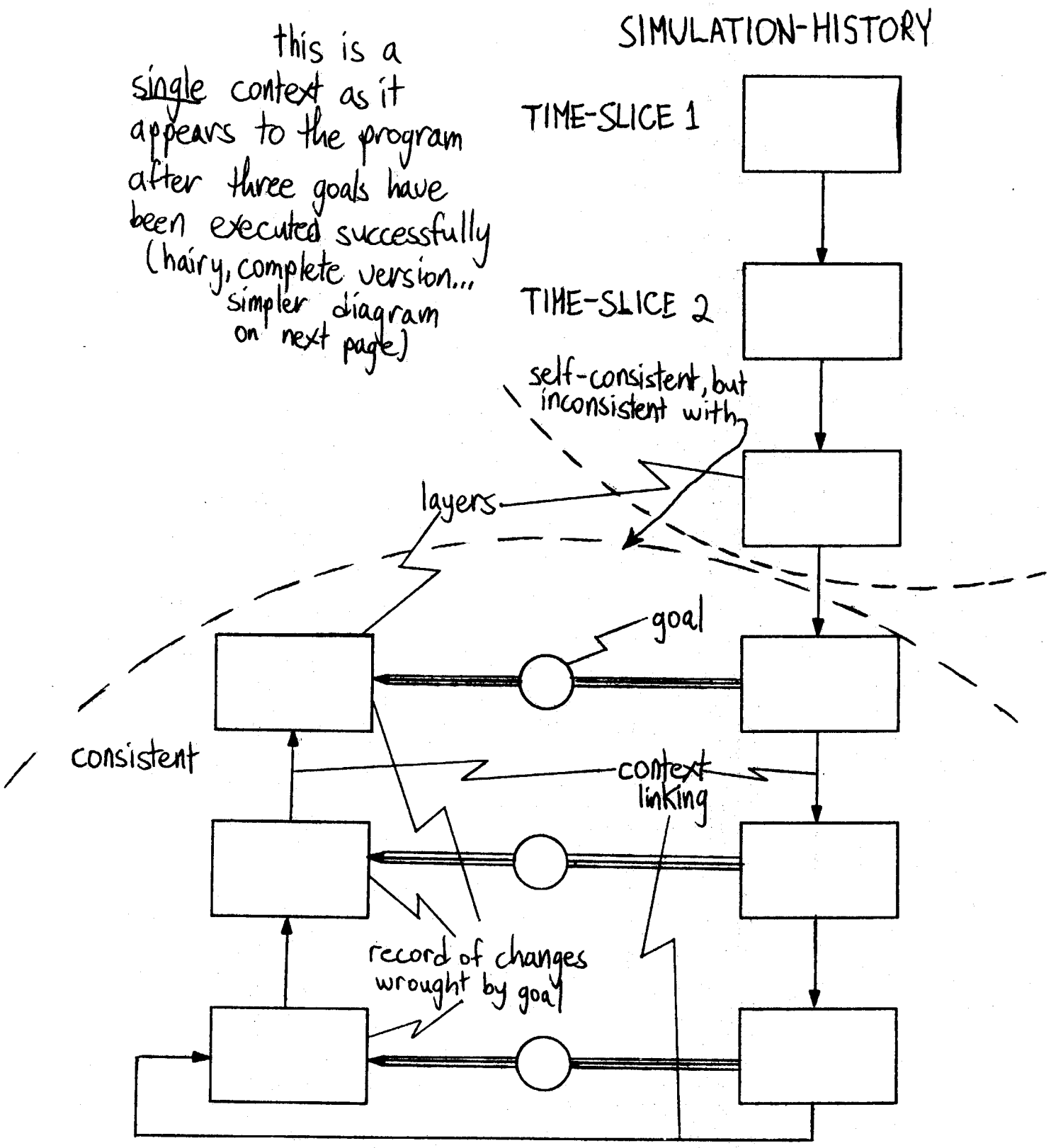
(1) They must, in fact, create a new simulation history.

records the changes made by a *GOAL to the particular TIME-SLICE involved. It then appends this new layer to SIMULATION-HISTORY and uses this new augmented context as the working environment of the debugger. Now, remembering the little discussion of context semantics in 4.2 (or, referring to [20]), we see that this causes the following effects:

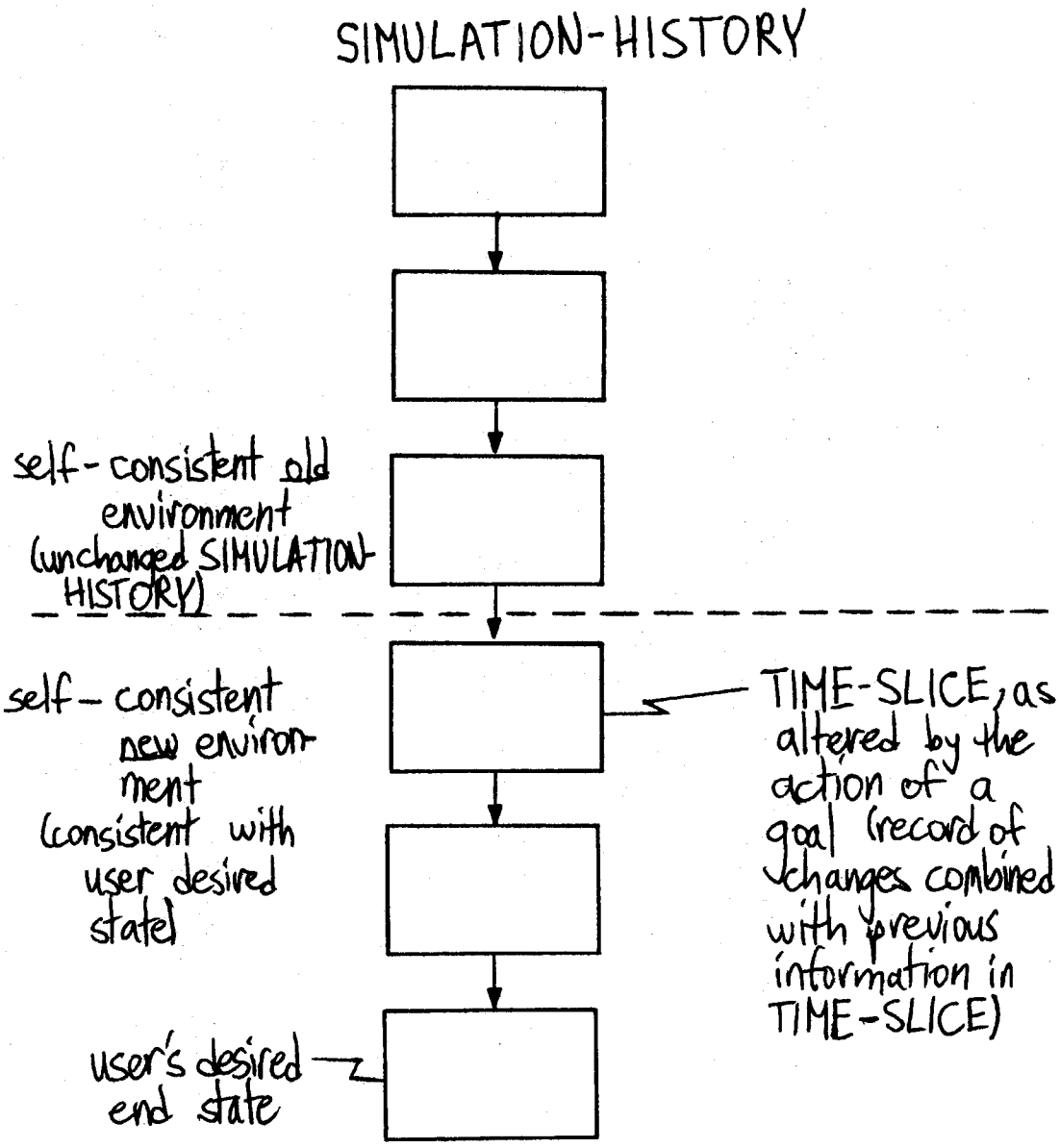
(1) The effects of a *GOAL are certainly localized since they occur only in a single layer which corresponds to a single TIME-SLICE.

(2) The debugger can always see a consistent environment by looking up the augmented SIMULATION-HISTORY as far as the last affected TIME-SLICE; the semantics of context then say that the data seen by the debugger is just what was in SIMULATION-HISTORY before (which is consistent via the simulator) except where contradicted by the parts that were changed by *GOAL's (which are consistent (up to that point) via the deductive mechanisms).

Perhaps it is well to interrupt here with an explanatory diagram...


this is a single context as it appears to the program after three goals have been executed successfully (hairy, complete version... simpler diagram on next page)
SIMULATION-HISTORY
TIME-SLICE 1
TIME-SLICE 2
self-consistent, but inconsistent with
layers
goal
consistent
context linking
record of changes wrought by goal

which is, due to the semantics of <u>context</u>, equivalent to:



which is certainly an easier conceptualization of what has gone on so far. However, the first picture is necessary to explain

(3) The layers which record the changes made by a *GOAL (the dashed parts of the first picture) can be peeled off and saved at any time, thus restoring the context to its original condition and saving the effects of the *GOAL (the track toward falure) for further use

This methodology fills the bill so far. Unfortunately, there is one final problem which complicates this little picture (you just knew there would be).

This complication comes from an as yet unseen aspect of the problem-solver: multiple goals. I mentioned earlier (section 3) the existence of "higher-order constraint interdependencies" in the model. (This weird-sounding effect was conveniently kept out of the example in section 2.) We will see in section 4.4.2.3 that higher-order interdependency leads to multiple goals. That is, instead of simple goals, the program must deal with constructs like:

```
(*GOAL (*AND
          (*GOAL ...)
          (*GOAL ...)
          (*GOAL ...)))
```

and

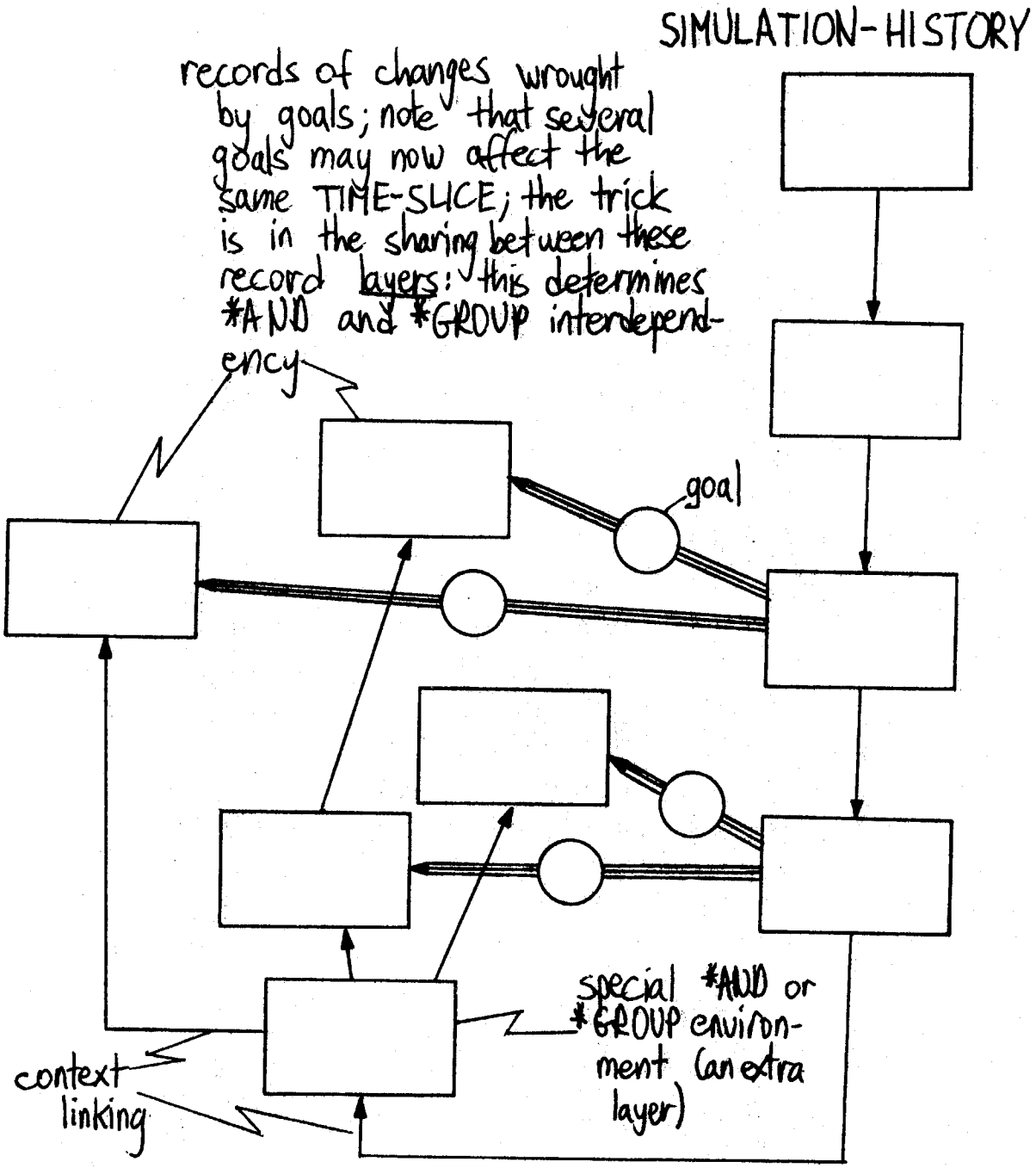
```
(*GOAL (*GROUP
          (*GOAL ...)
          (*GOAL ...)
          (*GOAL ...)))
```

We'll see more about multiple goals later. For now we need only examine one aspect of their behavior.

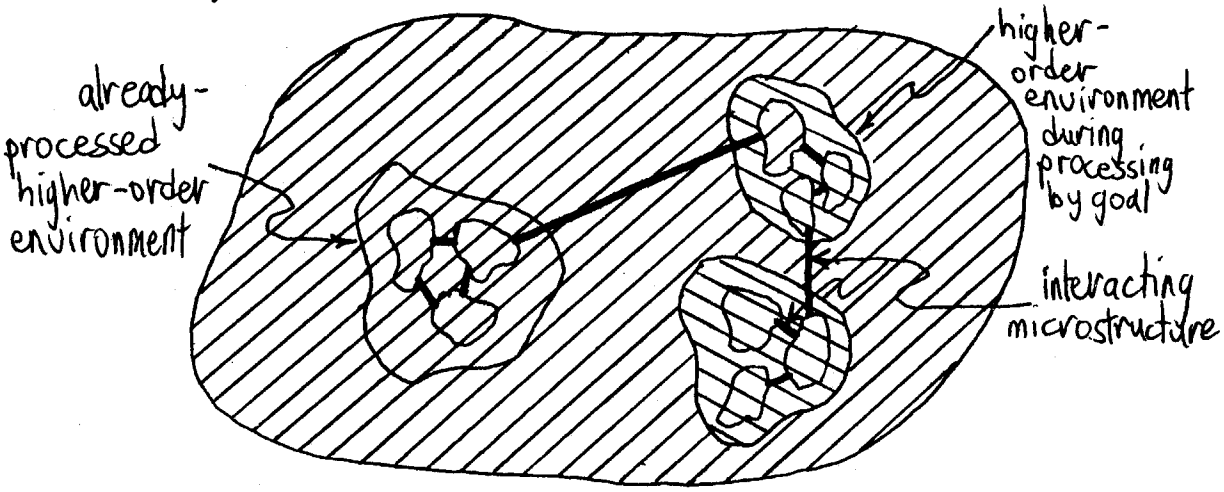
The raison d'etre of *AND and *GROUP is the expression of the fact that their component *GOAL's are not independent. That is, the *GOAL's they contain share common resources and cannot be achieved at each other's expense. (This is how they model interdependency.) Thus, the notion of a "local constraint environment" varies from the one bandied about earlier. Here we must have several *GOAL's sharing a single local environment. Furthermore, because of the interdependence of the *GOAL's, a component *GOAL that has not yet been completed must "see" the constraints posed by the completion of other component *GOAL's. Thus, the local constraint environment might cover several TIME-SLICE's.

Clearly this hairs things up a bit. Nonetheless, the program must preserve the semantics of these constructs because they are important effects of the model which give rise to their own special bugs (see 4.4.2.3). Actually, given the flexibility of contexts, the

implementation is rather straightforward. The little schematic of environments now looks like:



In terms of the previous discussion of perturbations, local and global environments, etc., nothing has changed except that the "local" environments now may have a hairy microstructure of local environments:



Uninterested readers may squint at the above picture (and concept), leaving everything as before.

Thus, a *GOAL indicates a local perturbation. The deductive mechanisms of the problem-solver follow through the interactions defined by the model to carry the perturbation throughout the simulation history in order to produce a consistent environment. The next section considers these deductive mechanisms and their interaction (via failure) with the bug-finders.

4.4 Debugging by problem-solving

The basic task of the program is to trace a bug from its manifestation to its source. This is done by taking in the manifestation as a *GOAL to be achieved (as discussed earlier). The process of achieving such a *GOAL is usually called "problem-solving". But this is a rather special use of problem-solving: the program expects to fail in the attempt. In fact, it is not until after a line of attack has failed that it becomes interesting to the debugger. In this section we see how lines of attack are formed, how they fail, and how they are used after they fail.

The most important part of any problem-solving process is the formation of subgoals (1) . Section 4.4.1 considers the methods (those deductive mechanisms we've heard so much about) for devising new subgoals in order to achieve a goal. This corresponds to asking the "how could we do this ?" question of section 2. But in this program, the object of the problem-solver is not this direct attack on the problem. Instead, the problem-solver must make certain it does not change the intent of the user's model in trying to debug it.

Thus, the process of attacking the

---

(1) Especially in this problem-solver. Since subgoals are rarely achieved, the whole process turns into subgoal-formation.

user's goal leads directly into the problem of separating the constraints which are in the simulation history because of user intent from those which are artifacts of unintended model operation. At certain key points in the deduction process, the program determines whether or not it should (in terms of user intentions) make the changes required by the deduction. This process of assigning GOOD and BAD REASON's to model action corresponds to asking the "why didn't you do this before?" question of section 2. In 4.4.2 we examine this REASONing process in terms of the philosophy of bugs presented in section 3.

The REASONing process leaves the program with a failed line of attack. This appears as a stream of *GOAL's, annotated at each point with the BAD REASON that triggered further program action. The program must then examine the record of the problem-solver to attach blame to the proper offending model part; i.e., to find the bug. This task of post-mortem recrimination is the subject of 4.4.3.

4.4.1 The attack

Here we examine the problem-solving phase of the debugging process. The key problem-solving

task of the program is to find the proper local changes throughout the global environment which will lead to the desired change. Since each desired change is represented by a *GOAL, the problem-solver proceeds by subgoal formation.

The subgoal-formation parts of the program (the "deductive mechanisms" mentioned earlier) are responsible for figuring out how one local change can be brought about by another. As an example of the way this cause-effect knowledge is procedurally represented in the problem-solver, the INCREASE function is presented here. The explanation of how INCREASE works will lead us directly into the REASONing methods of 4.4.2.

The program's main vehicle for asking the "how?" question is the INCREASE *GOAL:

(*GOAL (INCREASE <resource variable or submodel>
<amount> <time-slice> (1)))

That is, "goal: increase the resource variable or submodel by the specified amount in the specified time-slice." The user's initial *GOAL is usually of the INCREASE type (see section 2). This just means that the user's discrepancy is usually a defficiency of some resource variable (or lack of

---

(1) If a <time-slice> is not given, the program heuristically chooses one.

the appearance of some submodel) which he is asking the program to fix up.

As we saw in section 4.3, the program immediately sets up a hypothetical local environment in which the defficiency has been rectified. Then it tries to deduce an earlier environment which would cause the new desired simulation state. It does this deduction via the "logic of INCREASE" mentioned in section 2. The "logic", briefly stated, runs as follows:

(1) Constant quantities cannot be INCREASE'd

(2) In order to INCREASE a quantity that is a resource variable which is *OUTPUT (*REMOVE'd) by an *ACTIVITY or *EVENT, set up a new *GOAL to INCREASE (DECREASE) the number of occurences of that *ACTIVITY or *EVENT

(3) In order to INCREASE a quantity that is *RETURN'ed by a *FUNCTION, set up a new INCREASE-FUNCTION *GOAL (1)

(4) In order to INCREASE the number of occurences of an *ACTIVITY, set up (if necessary (2)) a new *GOAL to

---

(1) INCREASE-FUNCTION's major claim to fame is that it sets up *GROUP *GOAL's. I will therefore discuss it when I talk about *GROUP in 4.4.2.3 rather than here. For now it's okay to view INCREASE-FUNCTION as analogous to INCREASE applied to *ACTIVITY's.

INCREASE the resources needed by that *ACTIVITY

(5) In order to INCREASE the number of occurences of an *EVENT, set up a new *GOAL to INCREASE the frequency with which its *CONDITIONS are valid (which might include a *GOAL to INCREASE the number of occurences of the *ACTIVITY's which the *EVENT affects)

Clearly, the intent of this list is to cover anything which the user or another part of the program (1) might ask to INCREASE. However, the rules in the list are by no means of uniform character; they differ greatly in their logical bases.

The first rule can be viewed as a "fact", or, if you will, a property of the concept "increase." That is, the first rule depends only on the concept of "increase"--not on MSL, models, etc. The second rule expresses a definite property of MSL rooted in the semantics of *OUTPUT. It therefore depends not only on "increase", but also on the definition of MSL. The third rule, which will be discussed later, depends on "increase", the definition of MSL, and the rules of mathematics (since

---

(2) Some necessary resources may already be present in sufficient quantity.

(1) Since INCREASE is defined recursively, the "other part of the program" might be INCREASE itself.

mathematical functions are being increased). Again, it is valid for any MSL model. The fourth and fifth rules are different in a very important way. They depend not only on the definition of MSL and other "givens", but also on the particular model defined by the user.

The reason for this is that the occurence of *ACTIVITY's (and thus *EVENT's via the *ACTIVITIES construct (see 4.1)) can be directly determined by user intentions. These intentions are expressed by the *SCHEDULE modifier (see 4.1). *SCHEDULE is used whenever the modeller wishes to override the "always schedule when possible" default of the simulator. It therefore determines the pattern of *ACTIVITY and *EVENT activation throughout the simulation. *SCHEDULE is thus the primary expression of the user's policy for directing the dynamics of his model.

The fact that the "logic of INCREASE" must take into account user intention provides the key link between the "how?" and "why not?" questions. In the case of the first three rules of INCREASE, the "how?" question is perfectly well-formed. The program need only look at what is to be INCREASE'd without worrying about reasons <u>why</u> it shouldn't be done. There <u>are</u> no reasons, because the rules are valid for any case the program can encounter. Thus, the program can always go ahead and try the INCREASE. It

can either fail (1) (as in the case of INCREASing a constant, for example) or it can set up the next subgoal (usually another INCREASE *GOAL)--all without worrying about "should" and "shouldn't".

On the other hand, rules (4) and (5) must worry about "should" and "shouldn't" before setting up the next subgoal. Perhaps the user does not intend for the INCREASE to take place. Thus, INCREASE must ask the "why not?" question before it proceeds.

4.4.2 The voice of REASON

We saw in the previous section that the use of INCREASE to ask the "how?" question leads directly to the need for the "why not?" question. As usual, the program frames this question as a *GOAL. That is, given the *GOAL of INCREASing an *ACTIVITY "A" by "m" occurences in TIME-SLICE "n":

(*GOAL (INCREASE A m n))

---

(1) A failure of this kind is automatically for a "GOOD REASON"--see sections 2 and 4.4.2.1.

the program immediately forms the *GOAL

(*GOAL (SCHEDULE m A n))

to ascertain whether or not INCREASE should proceed.

SCHEDULE's job is to examine SIMULATION-HISTORY and the user's model to determine why the change suggested by INCREASE was not originally part of SIMULATION-HISTORY. After all, since it presumably leads to the desired state, why didn't the user cause the state suggested by INCREASE in the first place?

There are two kinds of reasons for the user's not causing the suggested state to occur initially. A GOOD REASON is that he deliberately intends (for reasons best known to himself) the model not to allow that state. A BAD REASON is that the interaction of the submodels has caused a constraint which disallows the state. A BAD REASON is <u>not</u> a bug. It simply implies that a constraint is due to submodel interaction and not user intention. However, given the bug philosophy of section 3, the program treats a BAD REASON as "suspicious"--a cause for further investigation.

In this section we examine the way the program distinguishes GOOD REASON's from BAD REASON's (and the way it classifies BAD REASON's). The next subsection discusses the program's model of user intent--i.e., its

method for discerning GOOD REASON's. After this, we classify BAD REASON's along the lines of the three "interaction bugs" presented in section 3.

4.4.2.1 GOOD REASON's

At each stage of the debugging process, the program is trying to change an environment...by using a resource, inserting a new submodel,etc. In order to do this, the program must face the question of whether or not the change should (in terms of user intentions) be made. Of course, it is unreasonable to expect the user to have to tell the program at each step what should and should not be changed. In fact, given the philosophy of section 3, it is very unlikely that the user could provide this information if he wanted to. Thus, the program needs some sort of theory of which of the constraints found in SIMULATION-HISTORY are user-intended and which are there because of a possible bug in the model.

Going back to sections 1.3.1 and 3, we recall the previous assumptions about user intentions: the user has a good understanding of each submodel, but only a very weak understanding of how submodels interact to achieve an overall goal. Thus, the program can assume, at least

temporarily, that all information in the simulation history which is derived directly from user statements about an individual submodel is user-intended. All other information is necessarily the result of submodel interaction and is therefore suspect. The programming task is to interpret this simple theory (1) of user intention in terms of the deductive mechanisms and SIMULATION-HISTORY.

Everything in an MSL specification pertains only to a specific submodel; this, in fact, was a design criterion (see 4.1). Thus, everything so far is user-intended, by our principle of locality. But this is only static information. Once the model is simulated, some of this static local information gives rise to interaction between submodels. The question then becomes one of determing how locality is preserved in the dynamic behavior of the model. That is, what's local about SIMULATION-HISTORY?

According to 4.3, the answer seems to be that the TIME-SLICE is used by the program as a "local

---

(1) This theory is of course quite liberal in its suggestion of "suspect" constraints. At this stage, this seems to be the best strategy. The deductive mechanisms are capable of eliminating non-bugs rather easily so that things don't blow up (see section 2). However, if really large models were used, a better theory would be necessary to avoid smothering the program with possible leads (see section 4.5).

environment"...but why? The TIME-SLICE preserves locality because direct user policy is at the TIME-SLICE level. Scheduling decisions set certain *ACTIVITY's to occur in certain TIME-SLICE's (see description of *SCHEDULE in 4.1). *PREREQUISITES are checked at the TIME-SLICE level, *OUTPUT occurs at the TIME-SLICE level, *FUNCTION's are called, *EVENT's triggered,etc.--all at the TIME-SLICE level. All of the direct user decisions, as specified by the static information in the MSL, affect the simulation at the TIME-SLICE level. Therefore, the program takes a constraint to be local (and thus user-intended) if it depends only on what happens in a single TIME-SLICE.

Now I mentioned in 1.3.1 that the models used in this thesis are especially interactive. Furthermore, as I said above, the criteria for suggesting unintended constraints can afford to be liberal--we would rather suggest wrong bugs than miss a possible bug. Thus, we would expect there to be few local "user-intended" constraints and many non-local "suspect" constraints. This is indeed the case. The resources present in any TIME-SLICE are dependent on the action of the model over many TIME-SLICE's and are thus non-local. Similarly, the timing of *ACTIVITY's which do not contain *SCHEDULE specifications becomes resource-dependent and thus non-local. *EVENT

occurences are specified by probabilistic functions of resources and are thus non-local. Finally, higher-order constraints like coincident presence of several resources span several TIME-SLICE's (see 4.3) and are, almost by definition, non-local. These non-local constraints give rise to the BAD-REASON's discussed in the next two subsections. For now, let's mention the few GOOD REASON's that exist.

Most GOOD REASON's concern constraints that arise from *SCHEDULE constructs. If the change requested by INCREASE would violate the *ACTIVITY's *SCHEDULE for that TIME-SLICE, SCHEDULE denies the request for GOOD-REASON (1) . Thus, if, as in section 2, there are three ADVERTISING *ACTIVITY's already in a TIME-SLICE and ADVERTISING contains the modifier

(*SCHEDULE 3)

SCHEDULE will deny any request to up the amount of ADVERTISING in that TIME-SLICE. Similarly, SCHEDULE views the other avatars of *SCHEDULE (see 4.1) as GOOD-REASON-generators.

The other kinds of GOOD REASON's are

---

(1) There is one exception to this which will be discussed in the next subsection.

those that are based on "fact" or are "true by definition" (see the first three rules of INCREASE in 4.4.1). Thus, SCHEDULE will deny attempts to schedule in negative time, increase constants, etc. for GOOD REASON. Actually, these REASON's can be viewed as being based on the "common sense knowledge" the user has in addition to his knowledge about submodels. That is, the user directly intends his model to be "sensible" as well as to be in accordance with known submodel constraints.

Thus, GOOD REASON's apply to constraints which depend only on single TIME-SLICE information, i.e., which reflect the locality which is characteristic of user intention. We now go on to investigate the way in which the program deals with non-local constraints.

4.4.2.2 Basic BAD REASON's

If the program cannot find a GOOD REASON for a constraint, it must attribute its existence to a BAD REASON. From the "interaction bug" philosophy of section 3 we see that the user's understanding of his model falters in the three critical areas mentioned at the beginning of this section:

(1) the effects of resource competition among submodels

(2) timing effects of submodels

(3) the effects of higher-order constraints

If a constraint is there for no GOOD REASON, the program considers the possibility that the constraint arose unintentionally from one of these three misunderstandings. It will therefore try to come up with a BAD REASON for the constraint's existence so that it can inform the debugger of the possible anomaly (see section 4.4.3). This section will consider the BAD REASON's related to the first two kinds of interaction. These BAD REASON's form the basis for BAD REASON's arising from higher-order interdependencies--as discussed in 4.4.2.3. Now, to continue with our favorite process, the SCHEDULE *GOAL was just seeing why the desired *ACTIVITY wasn't scheduled in that TIME-SLICE in the first place...

Since the user didn't specifically ask for the *ACTIVITY not to be scheduled, there can be only two reasons why the *ACTIVITY wasn't there:

(1) some of its prerequisite resources weren't present

(2) It is dependent on an *EVENT that didn't occur

Thus, the program first checks out the resource situation in the TIME-SLICE. If the resources are not sufficient to support the *ACTIVITY, there can be two reasons why:

(1) the resources were available in the TIME-SLICE but were used-up by higher-priority *ACTIVITY's before the *ACTIVITY in question got a chance at them

(2) the resources just ain't there

To check out the first possibility, the program looks at the status of the higher-priority *ACTIVITY's in the TIME-SLICE. If any of these *ACTIVITY's indeed "stole" resources which would have allowed scheduling of the desired *ACTIVITY, their names are collected and the BAD REASON

(PRIORITY-RESOURCE-BOUND (<names of offending *ACTIVITY's>))

is recorded.

If no higher-priority *ACTIVITY's stole the resources, then the resources must just have been absent from the TIME-SLICE in the first place. The ubiquitous two possible reasons:

(1) The *ACTIVITY's which *OUTPUT the desired resources weren't scheduled until it was too late for the resources to be available in the TIME-SLICE

(2) The *ACTIVITY's which *OUTPUT the desired resources were scheduled too early and the resources were gobbled up by higher-priority *ACTIVITY's in the intervening TIME-SLICE's

Of course, in either instance, the user may have intended this to be the case (well we know how to check that out...). On the other hand, the *OUTPUT *ACTIVITY's may have ended up in the wrong place because of the user's poor understanding of timing effects (1) --a BAD REASON. To determine which is the case, the program proceeds as follows. It first finds out what *ACTIVITY's *OUTPUT the desired resources and checks to see if they were scheduled too late to do the desired *ACTIVITY any good. Then, it sees whether the *OUTPUT *ACTIVITY's were "late" for GOOD REASON. If not, it notes a BAD REASON:

---

(1) Note that the "interaction information" about timing is implicit in the resources. That is, there are no explicit timer-alarms to say when something is too late or too early. The only evidence of a timing error in the model will be found in the levels of particular resources over time.

(RESOURCE-BOUND (TOO LATE (<names of offending *ACTIVITY's>)))

If there are no "late" *ACTIVITY's, or if the *ACTIVITY's were late for GOOD REASON, the program looks back up the SIMULATION-HISTORY for two things: *ACTIVITY's which *OUTPUT the needed resources scheduled "too early" for no GOOD REASON and "interloping" *ACTIVITY's of higher priority which stole the needed resources. If both of these things exist, the program notes:

(RESOURCE-BOUND (TOO-EARLY (<names of offending *ACTIVITY's) (<names of interloping *ACTIVITY's>)))

Thus, the PRIORITY-RESOURCE-BOUND and RESOURCE-BOUND BAD REASON's take care of the case in which the *ACTIVITY cannot be scheduled because of a lack of prerequisite resources (1) . This leaves the other case in which the *ACTIVITY could not be scheduled because it is

---

(1) As discussed previously, the program would try to alleviate this defficiency with an appropriate INCREASE *GOAL. The reason for this is to make sure that the program traces through the entire interaction path: after all, this resource defficiency could just be the result of an earlier decision which reflects the actual bug. More on this in 4.4.3.

dependent on an *EVENT that didn't occur.

The program can easily recognize this second case because it can only arise from the

(*SCHEDULE (ON <*EVENT-name>))

specification (see 4.1). If the specified *EVENT did not occur in the TIME-SLICE, the desired *ACTIVITY could not be scheduled. Now, if the program were acting like it did before, it would try to find out "why" the *EVENT didn't take place in the TIME-SLICE. However, this is inappropriate for *EVENT's, which, after all, model occurences which are beyond the modeller's direct control. Of course, this raises the question of why a modeller would make an *ACTIVITY dependent on an *EVENT in the first place. Indeed, the program becomes suspicious: it is possible that because of the user's poor understanding of timing effects, the *EVENT dependency (plus the time needed by the *ACTIVITY) will cause the *ACTIVITY to take effect at the wrong time--usually too late (1) . The program checks out

---

(1) The most common cause of this *EVENT-dependency is the "fire-fighting" approach to solving problems: when the event occurs, start doing something about it. (This is, in fact, the problem in the example of section 2: HIRING is dependent on QUITTING.) Note that this BAD REASON is the exception to the "if *SCHEDULE says it's okay, it's okay" dictum referred to earlier.

this possibility by looking up and down SIMULATION-HISTORY to see if the *ACTIVITY was scheduled "too late" or "too early". If either of these is the case, the program notes a BAD REASON:

(*EVENT-TRIGGERED-SCHEDULE <offending *ACTIVITY>
<"TOO LATE" or "TOO EARLY">)

If neither of these is the case, the program simply terminates its line of attack (1) on

(*EVENT-TRIGGERED-SCHEDULE)

and goes away mumbling to itself (actually, this would be the first "GOOD REASON" it looks at after all the BAD REASON's were checked by the debugger).

Well, this wraps up the "basic BAD REASON's" arising from poor understanding of resource conflict and timing effects. Now we go on to see how misunderstanding of higher-order constraints leads to the use of these same BAD REASON's in an expanded context.

---

(1) Note that unlike the other BAD REASON's, this one causes the line of attack to terminate--no further investigation is possible (see 4.4.3).

4.4.2.3 Higher-order BAD REASON's

Up until now (except for part of 4.3), I have over-simplified the interactive behavior of submodels for the purposes of discussion. Specifically, I have pretended that a submodel can depend on only one other submodel for its sources of input. Thus, my *ACTIVITY's have had only one unfilled *PREREQUISITE, my *FUNCTION's only one *ARGUMENT. This is of course quite unrealistic, and not a real restriction of MSL. In this section I remove this artificial restriction.

The introduction of multiple dependency brings up the issue of higher-order constraints. As we saw in 4.3, when submodels depend on several other submodels for input, the problem-solver must take into account the interrelationship of the input *ACTIVITY's. The input *ACTIVITY's are in fact operating under a "higher-order constraint" (see section 3.2)--they must combine to provide resources for a single *ACTIVITY (or *FUNCTION) at a certain time . This higher-order constraint is modelled by forcing the input *ACTIVITY's to share a local constraint environment (see 4.3). That is, all *ACTIVITY's sharing a higher-order constraint must be scheduled not only in accordance with their own needs, but also with the needs of

the *ACTIVITY or *FUNCTION that depends on them. There are two types of environment-sharing, reflected by two types of *GOAL's to handle the higher-order dependencies. The first of these is *AND, the expression of the way *ACTIVITY's depend on each other when their higher-order constraint is another *ACTIVITY. The second is *GROUP, which models the *ACTIVITY-*FUNCTION dependency.

*AND dependency arises from *ACTIVITY's that look like

```
(*ACTIVITY SALES-CALL
          (*PREREQUISITES
                    (*AND
                                        (*PRESENT (1000 CASH))
                                        (*PRESENT (1 UNIT))
                                        (*PRESENT          (SOME
SALESMAN))))
                .
                .
                .  )
```

That is, SALES-CALL depends on the submodels which *OUTPUT CASH,UNIT, and SALESMAN. All of these *OUTPUT's must be present at once (i.e., in the same TIME-SLICE). Thus, any *GOAL which tries to schedule a new SALES-CALL *ACTIVITY must take this into account. Specifically, if the resources are not available, all of the *OUTPUT *ACTIVITY's involved must be scheduled. That is, given the *GOAL

(*GOAL (INCREASE SALES-CALL m n))

and assuming none of the necessary resources are on hand (1), the program must generate the subgoal

```
(*GOAL
        (*AND
                (*GOAL (INCREASE CASH j n))
                (*GOAL (INCREASE UNIT k n))
                (*GOAL (INCREASE SALESMAN 1 n))
        ))
```

Now, just as before, the program must be careful not to INCREASE things contrary to the intentions of the user. Again, it uses the SCHEDULE *GOAL to find out the REASON for constraints. However, the SCHEDULE *GOAL cannot simply check out each INCREASE *GOAL independently as before. The INCREASE *GOAL's are now interdependent and must be treated as such. So now, finding GOOD and BAD REASON's is a whole new game.

Not really. Fortunately, the process isn't very different, especially in the case of *AND. First of all, examination of the whole GOOD REASON-finding philosophy and implementation will show that it is completely unaffected by higher-order interdependencies. This is almost by definition: GOOD REASON's pertain to individual submodels and TIME-SLICE's, while higher-order

---

(1) In section 2 I kept higher-order constraints out of the picture by buffering away dependencies. Thus, in the case of SALES-CALL, all resources except SALESMAN were available already (see section 2).

interdependencies transcend these boundaries of locality. Thus, SCHEDULE's GOOD REASONing processes are still the same. Certainly, however, the BAD REASONing is different. But most of the differences have been taken care of already by the environment-sharing discussed in 4.3. That is, the effects of higher-order constraints on resource conflicts and time dependencies are already reflected in the way *AND *GOAL's are set up and processed--the higher-order interdependency is already modelled. For example, if satisfying one component *GOAL steals resources from another or disturbs the timing of another, the shared environment will make this interaction explicit: the resources needed by each *GOAL are recorded separately so that the effects of everything done in the *AND environment can be traced to the proper source.

All this is saying that all SCHEDULE has to do about *AND's is to realize that it is in a shared environment and attribute BAD REASON's to the effects of sharing. Thus, the searches for higher-priority *ACTIVITY's and timing problems which were previously carried out only in a single TIME-SLICE are now carried out in the whole *AND environment. The "new" BAD REASON's they generate look like

(PRIORITY-RESOURCE-BOUND (<names of offending *ACTIVITY's>) *AND-MODE)

(RESOURCE-BOUND (TOO-EARLY (<names of offending *ACTIVITY's>)

*AND-MODE (<names of interloping *ACTIVITY's in the *AND environment>) (<names of other interloping *ACTIVITY's>))

etc.

The theme here is that most of the work for finding higher-order BAD-REASON's in the *AND case was done by setting up the *AND environment in the first place. That is, the interdependency is already explicitly modelled by the way *AND *GOAL's work, and need only be checked through by SCHEDULE to find the appropriate BAD REASON's. This theme is elaborated for the *GROUP case.

In 4.4.1 I postponed the issue of INCREASing *FUNCTION's by attributing this task to a separate INCREASE-FUNCTION *GOAL-type. The job of INCREASE-FUNCTION is to figure out a way to increase the value *RETURN'ed by a *FUNCTION by changing the values of its *ARGUMENTS (thus, it is completely analogous to INCREASE). Obviously, this problem is extremely difficult for a large class of functions. Fortunately, the functions needed in business games, and, indeed, in most of business processing, are of a very simple nature (1) . MSL currently

---

(1) The mathematics of management <u>science</u>--i.e., mathematics meant to model systems and decisions--can be quite sophisticated, but this is not business processing. Indeed, even in a business game, the probability-handling can get tricky. But all of this is built into MSL--the user can

allows the representation of only two kinds of functional dependencies: tables and linear functions of a few variables. The mathematical techniques for increasing these *FUNCTION's are simple and are not of interest here. The interesting part of *FUNCTION's for this discussion is they are responsible for the second kind of higher-order interdependency.

We just saw how the relation between *PREREQUISITES and *OUTPUT's causes *AND interdependency. Similarly, the relation between *ARGUMENTS and *RETURN'ed value causes *GROUP interdepency. In the *AND case, the interdependency was that all *PREREQUISITES must be present in the proper quantities in a single TIME-SLICE for the *ACTIVITY to be initiated. *GROUP interdependency is weaker. We know only that some combination of changes to the components will bring about the desired change to the higher-order constraint. That is, each subgoal can contribute an unspecified amount to the success of the overall *GOAL. Perhaps the increase of only one of the *ARGUMENT resources will suffice to increase the *RETURN'ed value. Or, all may be necessary--making the *GROUP an *AND at the extreme.

Now the program must model this kind of

---

only define simple functions which use the probability machinery.

interdependency when it tries to INCREASE *FUNCTION's. Furthermore, in trying to solve the INCREASE-FUNCTION problem, it must go about the task pretty much the same way organizations do in order to run into the same kind of interactive behavior. That is, the interaction involved in a kind of breadth-first approach to the problem (increase each *ARGUMENT resource a little in turn until the *RETURN'ed value has been INCREASE'd the desired amount) causes very different subgoal interaction than, say, a depth-first approach (increase each *ARGUMENT as much as possible separately to see how much it helps to INCREASE the *FUNCTION). The differences are in which subgoals are allowed to be achieved at the expense of others (1) , the range of subgoals tried, and the extent to which each subgoal is exercised (2) . Clearly, different interdependencies are tapped by different subgoal attack methods.

So the program must try to overcome the

---

(1) Unlike *AND, this is allowed because not all *GROUP'ed subgoals must be achieved. The only requirement is that all of the subgoals which eventually succeed must share the same local constraint environment (otherwise the construct doesn't model higher-order interdependency).

(2) Note that this need to model the organization's problem-solving method was not present in the *AND case. Since all subgoals must be achieved as stated, no "resource-stealing" is allowed among them and all of them must be fully tried and executed.

higher-order constraint of increasing a functionally-determined value the same way organizations do. Obviously, this is a tall order. First of all, functional relationships are usually implicit in organizations, not explicit as in MSL--so it's hard to see what organizations do about them. Second, it is reasonable to assume that different organizations attack different functional problems in different ways at different times. Finally, it is possible that the actual process is not pre-defined at all in many cases, but is instead made-up and modified during the course of each problem's solution. What I am trying to say by all of this is that I'm not about to solve the whole problem or even a very big part of it...

What I <u>have</u> done is to program a single, slightly sophisticated method of attacking higher-order functional constraints which attempts to model one way in which an organization might do it. It should be seen as an experiment for demonstrating the approach of the program in dealing with this kind of constraint, not a fully developed piece of the system. This part of the program, incorporated in INCREASE-FUNCTION, works as follows: given a *GOAL of the form

```
(*GOAL
        (*GROUP
                (*GOAL (INCREASE argument1 time1))
                (*GOAL (INCREASE argument2 time2))
                .
```

```
                    .
                    .
                    .
                ))
```

the program takes the first *GOAL

```
(*GOAL (INCREASE argument1 time1))
```

and tries to INCREASE argument1 the minimum possible amount as a "feasibility study". It carries the *GOAL all the way to completion, if it can. If the *GOAL is unsuccessful (for GOOD REASON), it is withdrwan from the *GROUP and the program does a "feasibility study" on the next *GOAL in the *GROUP. If no "feasibility study" is successful, the whole *GROUP naturally fails. Now, if any of the "studies" are successful, the program will keep attacking the studied line until it fails. When this happens, i.e., when the particular *ARGUMENT has been INCREASE'd as much as possible, the program considers itself to have a "partial success". That is, the effect of the INCREASE'd *ARGUMENT is now calculated into the overall *GROUP *GOAL, so that a new *GROUP *GOAL is formed such that

(1) The fully INCREASE'd *GOAL is no longer in the *GROUP

(2) The overall *GOAL is reduced by the amount contributed by the successfully INCREASE'd *GOAL

In this new *GROUP environment, the other *GOAL's are similarly processed until success (or failure) occurs.

All of this hopefully goes toward modelling the way an organization attacks this kind of problem: by checking out and eliminating possibilities one by one, and pushing winning lines as far as possible to achieve the overall *GOAL. As intimated in 4.3, the process is modelled (like *AND) by the proper sharing of environments. Obviously, the environment-hackery for *GROUP's is a bit more complicated than for *AND (for example, it must incorporate the notion of "partial success" and the fact that all the eventually successful *GOAL's and only the eventually successful *GOAL's share the same local constraint environment). The question for us here is how this affects the GOOD and BAD REASONing process.

Again, the answer is "not all that much". As with the *AND case, the only difference is that the BAD REASON's differentiate between constraints caused by higher-order interaction and those caused by other kinds of interaction. This is again just a matter of tracing through the explicit relationships set up in the *GOAL's environment structure. As far as actual BAD REASON's for constraints go, *GROUP only adds two (minor) new wrinkles. First of all, it will make a special notation if the constraint comes

up during a feasibility trial. Second, it carefully notes which *GROUP *GOAL's have already succeeded when the constraint comes up. These are just convenience factors which the bug-finder uses when suggesting *GROUP bugs to the user; it wants to make clear exactly what the program was doing when it ran into the constraint. This is important, because, as mentioned above, different interaction occurs depending on exactly what the program does.

This brings up a final important point. *GROUP BAD REASON's are perhaps the weakest in the REASON repertoire because they depend directly on the actual exploration methods used. That is, the program might suggest a BAD REASON which the user may never really encounter because of the way his organization handles functional dependencies. Thus, the debugger saves *GROUP-type bugs for last. Nonetheless, I think that it is very important to include this kind of REASONing in the debugger: *GROUP-style dependencies are pervasive in organizations. Furthermore, they point the way toward modelling more sophisticated kinds of submodel-submodel interactions . The weakness of the *GROUP method in this program is its incompleteness, not its basic concept.

This section has catalogued all of the BAD REASON's generated by the program. Now we finally get

around to finishing the bug story by showing how the BAD REASON's are used to suggest the actual model bugs.

4.4.3 The post-mortem recriminations

So far, the debugger has been left with a bunch of GOOD and BAD REASON's for constraints. It is now time to turn these into bug suggestions. So, let's see what the REASON's mean to the debugger. If the problem-solver is faced with a BAD REASON for a constraint, it knows that the constraint is based on submodel interaction. Its job is to explore that interaction. Therefore, when SCHEDULE returns a BAD REASON, the problem-solver considers it a cause for further investigation. In this way, it carries the perturbation as far as it can--tracing the interaction patterns to their roots.

GOOD REASON's are the "roots" that stop this search through the interaction path. They imply that the constraint blocking the path is not due to interaction, but rather to direct user intent. The program should not disturb user intent, since its only purpose in changing the environment is to debug the existing model. It now has a GOOD REASON to stop changing the environment, so it stops.

its current line of attack is said to "fail" (in its attempt to bring about the desired change). Thus, the problem-solver's activities leave a line of *GOAL's attached to BAD REASON's ending in a *GOAL attached to a GOOD REASON (1) . Now what does all of this have to do with debugging? Simply this: the program has now tried to overcome every interaction-based constraint in the way of producing the user's desired state. It has reached a user-desired constraint which is the root cause of all of the interaction-based constraints. Therefore, it has reached the end of the line and cannot produce the user's desired state. There can be three reasons for this state of affairs:

(1) The user's desired state is off-base: he has set the model an impossible task

(2) One of the user's original intentions is wrong; i.e., one of the root constraints is the bug

(3) One or more of the interaction-based constraints between the root constraints and the desired state are incorrect: the model has an interaction bug

It is obvious from what has been said before that the program thinks that possibility (3) is the most likely. It therefore suggests that one or more of the interactive constraints (noted by BAD REASON's) are caused by the bug.

---

(1) Except for the *EVENT-TRIGGERED-SCHEDULE case discussed in 4.4.2.3.

That is, given that the interaction constraints are wrongly causing the discrepancy, the debugger's job is to find the part of the model which gives rise to the faulty constraints. This is then suggested as the "bug" in the user's model. If the user doesn't agree with any of the program's suggestions based on possibility (3), the program falls back on (2), and finally (1). Anyway, let's pick up the process again at the possiblity (3) suggestion phase.

The program now has the location of the bug bracketed between the beginning and end of a "line of attack". Furthermore, the submodels which could have caused the bug have been narrowed down to a relatively smalll "interaction group" (the union of all submodels mentioned in the bracket) (1) . The program must now pick out the

---

(1) The size of the "bracket" and "interaction group" of course depends on the model. However, in the experience I have had, the relevant groups have been small: a few BAD REASON's and thus slightly more possible submodels. In the case of higher-order stuff, the group gets somewhat larger. There is no reason to expect brackets or interaction groups to get much larger for larger models: the key factor in determining their size is the amount of control the user exercises over his model (in MSL, the extent to which things are determined by *SCHEDULE's). Control means GOOD REASON's and thus short paths between initial manifestations of a discrepancy and GOOD REASON's to close the bracket. Control also means smaller groups of submodels which can affect the timing and resource-allocation of other submodels. Since managers (and modellers) exert considerable control over their systems, the amount of uncontrolled interaction possible in any realistic model is probably quite reasonable-sized. This in turn means that brackets and interaction groups should also stay reasonable-sized.

submodels in the "group" which caused the BAD constraints in the "bracket".

Sometimes this is quite easy: all of the BAD REASON's are traceable to a single submodel interaction. Examples of this are the *EVENT which triggers an *ACTIVITY at the wrong time, the *ACTIVITY which constantly steals resources from other necessary *ACTIVITY's, and the *ACTIVITY which is always too late (too early) to allow another *ACTIVITY to be initiated on time. The program looks for these single-cause interactions by scanning the BAD REASON's in the bracket, looking for "give-away" BAD REASON's like *EVENT-DEPENDENT-SCHEDULE or consistencies in the "offending *ACTIVITY's" and "interloping *ACTIVITY's" listings. If, in the process of examining the bracket, the debugger finds a single such cause for the BAD REASON's of the bracket, it immediately labels the faulty interaction (i.e., the submodels involved in the interaction) as the bug for that bracket, and files it away. Often, however, in looking at the BAD REASON's of a bracket, the program finds that a particular BAD REASON could have been caused by any of several interactions. For example, *ACTIVITY A couldn't be scheduled because B stole its resources, or because C caused D to be late so that D couldn't provide the necessary resources for A. The program handles this by noting each

cause separately as a bug.

Sometimes this straightforward process breaks down: the program is unable to pick out the cause for the BAD constraints of a bracket (this happens mostly in *AND's and (especially) *GROUP's). Currently, the program simply presents the troublesome bracket to the user telling him that "there's something wrong in there". I consider this an incomplete part of the program (see 4.5).

When the program has found the bug (or the few bugs) for each bracket, it presents them to the user in order of "likelihood". The debugger's model of the likelihood that a suggested bug is actually a bug in the model is

(1) The more specific the suggested bug, the more likely it is that it is genuine; thus, bugs like *EVENT-DEPENDENT-SCHEDULE which correspond to a single BAD interaction are suggested first.

(2) The more definite a suggested bug, the more likely it is; i.e., brackets which contain a single possible bug are suggested before those with multiple bugs, which are in turn before those which are just brackets with the "something's wrong" tag.

(3) The more interactions encompassed by a single bug, the more likely it is; this is just a recursive application of Murphy's law...the more interaction decisions a user has to make, the more he'll blow--thus *AND bugs (1) and long timing chain bugs (A was late for B was late for C was...) come early.

(4) Timing bugs are more likely than resource-conflict bugs; PRIORITY determinations are much closer to local specifications, and are thus more likely to be user-intended than the multi-TIME-SLICE machinations of a timing bug.

(5) *GROUP bugs are saved for last.

(6) After all of the bugs due to interaction are gone, the program works on the second possibility stated above--i.e., it starts suggesting that the GOOD constraints are faulty (i.e., wrong *SCHEDULE specification, etc.); it starts with the *EVENT-DEPENDENT-*SCHEDULE GOOD REASON if it's around--it's suspicious.

---

(1) *GROUP bugs would be here too, except, as I mentioned in 4.4.2.3, for the fact the mechanism for handling them is rather dubious.

(7) The program suggests missing submodels (see 4.5).

Thus, the program goes through its suggestion repertoire bug by bug, providing the user with an orderly statement of what the program thinks might be wrong with the model (see section 2 for the format of the suggestions). The user can always ask to see the interaction path leading to a bug, the bracket of a bug, and any other bugs which pertain to a particular bracket.

If the user does not agree with any of the bugs suggested, the program will suggest possibility (1): that his original *GOAL was wrong. If the user is still unsatisfied after all this work, the program informs him as to the location of his head and logs him out.

4.5 Don't confuse me with the facts

Most of the program's knowledge about models is contained in its conceptions of MSL (including, for example, its ideas of how to INCREASE MSL quantities) and its notions of user intention--as discussed in 4.4. However, as I mentioned in section 2, it is useful from a debugging point of view to include actual "world" knowledge of business games. Clearly, this knowledge can be used to suggest bugs which transcend the MSL specification.

This is, in fact, the only use the current program has for WOBG knowledge. As shown in section 2, the program has a facility for suggesting "missing" parts of an MSL specification. This comes from a (very simple) model of what an MSL model of a business game (1) could contain. The program simply checks at various points to see whether the addition of an *ACTIVITY could solve some problem (usually alleviate some defficiency) in the user's model. Thus, when there is a lack of CASH in the sample run in section 2, the program notes that the addition of a FACTORING *ACTIVITY (see description in Appendix A and specification in Appendix B) could solve the problem.

While this sort of thing is certainly useful, it is only a "zeroeth order" attempt at using world knowledge in debugging. A more important use of WOBG knowledge would be to aid in finding bugs within the MSL specification (i.e., the same kind of bugs the program now finds). As I mentioned in 4.4, a major determiner of the efficacy of the debugging program is the number and size of the "brackets" which enclose possible bugs. In the current program, brackets are determined by the amount of uncontrolled interaction--i.e., a purely MSL-level criterion. In a more thorough-going approach, WOBG

---

(1) In fact, it is based entirely on the game in Appendix A.

knowledge could be used to determine which interactions are really natural and which are possible bugs (1) --thus limiting or even eliminating brackets. Also, WOBG knowledge could be used to suggest suspiciously specified *ACTIVITY's, etc.

The main reason that I have not exploited WOBG knowledge in these more sophisticated ways is that it has not been necessary for the models I have investigated so far. Furthermore, it is interesting to see how far a "domain-independent" (2) debugger can go toward finding bugs in MSL models. Thus, WOBG knowledge does not enter into the main bug-finding process at all. Its sole use is in suggesting the addition of *ACTIVITY's to the current model (3) .

---

(1) This sort of thing is actually found to some degree in the programs of Sussman |18| and Goldstein |5|.

(2) See Sussman's discussion of the domain-independence of HACKER |18|.

(3) It operates off a WOBG database which will not be described here. It works a lot like MAPL |10|, and was in fact designed to be compatible with the larger MAPL database of Protosystem I (the WOB |9|).

5 Conclusions

I would like to use this concluding section to fit my model-debugging system into the "big picture", viewing it first as a debugging tool, and second as part of an automatic programming system.

The approach of my debugging system should be seen as one method of the several which can be used by the human or machine problem-solver. The simulate-and-investigate technique shown here is useful for debugging poorly understood but easily modelled systems. It requires the modeller's knowledge and lack of knowledge to be of a certain character, as outlined earlier. It is also most useful for handling highly interactive systems. If the problem domain is very well understood, or if actions in it are basically independent, other techniques are simpler and much better.

Furhtermore, it should be stressed that the debugging methods of the program are quite naive in the context of a real (i.e., non-game) interactive system. It is almost certain that all of the techniques described here would have to be shored up with procedures based on knowledge of the problem domain (see 4.5). Remember that

the basic "smarts" of my system is in the exploration of the simulation history. In real life, this exploration phase is usually preceded by some knowldgable guess work on the part of the debugger: almost all expert human debuggers (programmers, consultants,etc.) start their exploration for a bug with a good preconceived notion of the nature of the bug. This "notion" comes from the utilization of long experience about what kind of bugs are attached to what kind of problems; most debuggers know that only one or two things could possibly cause a bug at any given time in their exploration. No one yet knows how to encode this key experiential knowledge into a computer program. Certainly, no attempt has been made in this thesis.

Thus, the program presented here, when viewed only as a general debugging technique, should be seen as part of a larger system: it fits in after an initial "guesswork" phase (as one of several possibly applicable techniques) and just before a "weeding out" phase which makes thorough use of knowledge in the problem domain to narrow down the choice of possible bugs.

The model-debugging needs of an automatic programming system are somewhat different. Here the user is interested in expressing a model of his problem to the machine in such a way that he can be sure that the

machine understands it properly. Thus, after a phase of model specification aid at define-time (1), a model-debugging system like the one here can come in and demonstrate the APS's idea of the model to the user's satisfaction (and help the user overcome any dicrepancys). The simulate-and-investigate and domain-independence philosophies of my system are well-adapted to this purpose: the system can afford to be an expert in its own modelling language and do a great deal of exploration work in finding bugs. Furthermore, the user can tolerate a reasonable number of program-generated choices of bugs in his model if he can be certain of eventual understanding by the APS. Therefore, I think that the techniques used here might find direct application in automatic programming.

Nonetheless, for a debugger to be truly useful, whether in an automatic programming or general artificial intelligence environment, it must incorporate the same kind of experiential debugging knowledge found in the human expert. This kind of stuff will surely be the basis of the next generation of debuggers which are now on the horizon.

---

(1) See [9] for Protosystem I's "activity expert modules".

Bibliography

|1| Balzer, Robert, "Automatic Programming", Institute Technical Memorandum 1, University of Southern California Information Sciences Institute, Sept., 1972.

|2| Boehm, Barry W., "Software and its Impact: A Quantitative Assessment", RAND Study P-4997, Dec., 1972.

|3| Forrester, Jay W., Principles of Systems, Wright-Allen, Cambridge, Mass., 1968.

|4| Galbraith, Jay R., "Organization Design: An Information Processing View", unpublished Sloan School of Management working paper No. 425-69, MIT, Cambridge, Mass., Oct.,1969.

|5| Goldstein, Ira, Understanding Fixed Instruction Turtle Programs, PhD Thesis, MIT, Cambridge, Mass., Sept., 1973.

|6| Gorry, G.A., "The Development of Managerial Models", Sloan Management Review, Vol. 12, No. 2, Winter, 1971.

|7| Hewitt, Carl, Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot, PhD Thesis, MIT, Cambridge, Mass., April, 1972.

|8| Little, John D.C., "Models and Managers: The Concept of a Decision Calculus", unpublished Sloan School of Management working paper No. 403-69, MIT, Cambridge, Mass., June, 1969.

|9| Martin, William A., "Interactive Design in Proto-system I", Project MAC Automatic Programming Group Internal Memo No. 4, MIT, Cambridge, Mass., August, 1972.

|10| Martin, William A. and Rand B. Krumland, "MAPL, A Language for Describing Models of the World", Project MAC Automatic Programming Group Internal Memo No. 6, MIT, Cambridge, Mass., Oct., 1972.

|11| McKenney, James L., Simulation Gaming for Management Development, Harvard Division of Research, Boston, Mass., 1972.

|12| Minsky, Marvin L., "Matter, Mind, and Models", in Semantic Information Processing (Minsky,ed.), MIT Press, Cambridge, Mass., 1968, pp.425- 432.

|13| Reitman, Julian, Computer Simulation Applications, Wiley-Interscience, New York, N.Y., 1971.

|14| Rockart, John F., "Model-Based Systems Analysis--A Methodology and Case Study", unpublished Sloan School of Management working paper No. 415-69, MIT, Cambridge, Mass., Sept.,1969.

|15| Rustin, Randall (ed.), Debugging Techniques in Large Systems, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1971.

|16| Stogdill, Ralph M. (ed.), The Process of Model-Building in the Behavioral Sciences, Ohio State University Press, 1970.

|17| Sussman, Gerald J., "The FINDSPACE Problem", VISION FLASH No. 18, AI Lab Vision Group, MIT, Cambridge, Mass., Aug., 1972.

|18| Sussman, Gerald J., A Computational Model of Skill Acquisition, PhD Thesis, MIT, Cambridge, Mass., Aug., 1973.

|19| Sussman, Gerald J. and Drew V. McDermott, "Why Conniving Is better than PLANNING", AI Memo No. 255A, AI Lab, MIT, Cambridge, Mass., April, 1972.

|20| Sussman, Gerald J. and Drew V. McDermott, "The Conniver Reference Manual", AI Memo No. 259, AI Lab, MIT, Cambridge, Mass., May, 1972.

|21| Winograd, Terry, Procedures as a Representation for Data in a Computer Program for Understanding Natural Language, PhD Thesis, MIT, Cambridge, Mass., Feb., 1971.

Appendix A

The following is excerpted from the article "Business Games--Play One!" by G.R. Andlinger in the Harvard Business Review for March-April, 1958 (© The President and Fellows of Harvard University)--it is reprinted by permission.

It serves as an example of the kind of business games at which the program (and MSL) are directed. An MSL model of the game described here appears in Appendix B.

Business Games--Play One!

Basic Objectives

Games are as old as man. Usually, their basic objective is entertainment. The Business Management Game, however, aims not at entertainment, but at learning. Other differences between it and a game like Monopoly, for example, are:

--The degree to which it approaches reality.

--The degree to which the players' experience, judgment, and skill--as opposed to luck--influence the outcome.

If any business game is to serve a purpose beyond that of a fascinating toy , there must be some transfer of learning from the game situation to reality. While there probably is some such transfer from playing a generalized business game that mirrors "any company" and not a particular firm, an executive could derive infinitely greater benefit from a game that permits him to practice guiding the destiny of his own company or one in his own industry--which unfortunately, is unavailable at this early stage of business gaming. The success of specific war games, which the military has been using for years to simulate combat situations for training officers, however, holds great promise for similar applications in business in due course.

The Business Management Game is a case in point. We started it in 1956 with the idea of applying war-gaming techniques to business. In the course of the year we tested, modified, and retested the game many times to develop a fine balance between realism and playability. The more closely a game resembles reality, the more cumbersome it becomes--until it is no longer playable. Hence, there is a need to compromise. Also, we designed the game to be relatively stable. No extreme strategy can result in sudden success; yet players can gain outstanding success if they are good enough--or bankruptcy if they are not careful.

The game is partly deterministic and partly probabilistic. Some results are determined directly by the action of the players; others are, to varying degrees, subject to chance or probability. The weight of the elements of the game is such that the longer the game,

the smaller the influence of luck.

Rules of Play

In this section I shall give a brief general description of each game element and the specific values, rules and probabilities that define each element in quantitative terms. Instructions for the umpires are included at each point; but remember that they should not be given to the players.

The Market

The market is made up of 24 customers. Each customer's potential is different; in any one time period, a few customers are not buying any units, while others may buy four or five units (at $10,000 per unit) If a salesman is able to make a sale.

The market is dynamic, so the customer potentials change. If the market is growing, they change upward; should the market be hit by a recession, however, they may drop drastically. The long-term trend of the market is announced to the players; short term fluctuations are not. If a company is interested in finding out what the total market potential is in any time period, a $2000 expenditure for market research will buy this information from the umpires.

The 24 customers divide geographically into four regions on the game board, each region containing six accounts. This geographical division allows the company to do local advertising (see the section on "Advertising the Product") and conduct market research in only one region at a time. Such market research, which tells a company the potential of each customer in the region and permits the pinpointing of the direct selling effort (see the section on "Marketing the Product"), may be obtained by paying the umpires $30,000 for "staff work."

In addition to the separation into geographical regions, the market breaks down into one rural and two urban markets. The significance of this distinction is that in an urban market, where a salesman can make more calls per day, he has two chances of making a sale during each time period, while in the rural market he has only one chance.

If at the end of a year a company desires to find out what portion of the total market it has been able to capture, it may but a share-of-market

information from the umpires for $2000.

The umpire should:

(1) Keep a list of all current account potentials.

(2) Distribute a total customer potential, which comes to $360,000 at the beginning of the game, at random to the 24 customers as follows:

1 account	$40,000
3 accounts	30,000
5 accounts	20,000
13 accounts	10,000
2 accounts	0

(3) Depending on the economic climate determined in advance, change these starting potentials as the game progresses as follows:

--For slow growth, chane one account each quarter at random. Move ahead on the random number table until a number between 01 and 24 appears, then add $10,000 to the potential of that account number.

--For faster market growth, change two or three accounts in the same mannner as above for each quarter.

--For a depression, change half or all of the accounts to zero for one or more quarters.

(4) If a company decides to buy market information (total potential, market research, or share of market), write the information on a slip of paper and pass it to the company.

Marketing the Product

Units are sold by salesmen, who call on the 24 accounts in the market. In an urban market a salesman may make two calls per quarter; and in a rural market, only one.

In the presence of an umpire, the sales manager of a company points to the accounts he wants to call on. The umpire will tell him, after examining the random number table, whether a sale is made or not. How many

units are sold to a customer will depend on competitive action. The completed decision form, returned to the company at the end of the particular period, contains the actual sales results by accounts.

Whenever a salesman has two calls, he must make the second call on a any of the three to eight accounts adjacent to the first square called on; that is, he may not jump accross territories. If no sale is made on the first call, he may, of course, call on the same account again during the same quarter. Furthermore, there is no limit to the number of salesmen who may call on the same account in one time period. Between quarters, salesmen may be moved to any accounts that the company wishes to cover during the next quarter.

Each time a salesman makes a call, he has a certain fixed probability of making a sale. This chance of making a sale may be increased in one of three ways or a combination thereof:

--A company may intensify its direct selling effort by having more than one salesman cover one account as described above. In such a case, if the first salesman makes a sale, the second one may move to any adjoining account for his calls.

--A company may support the salesman's effort by advertising (see "Advertising the Product").

--A company may attempt to improve its product by spending more money for a research and development effort (see "Research and Development").

Every salesman costs $10,000 to hire and then $1000 per quarter in slary. (Since the product he will be selling is a high-price, complicated unit, it takes one year to train a salesman before he can be sent out into the field.) There is a possibility that a salesman will resign, in which case the umpire informs the company of this loss.

The umpire should have the following instructions for marketing:

(1) Each period there is a 5% chance of loss for each salesman. Move ahead on the random number table as many numbers as the company has salesmen; if one or more of these numbers is .05 or less, the company loses one or more salesmen.

(2) In an urban market, allow two calls per

quarter; in a rural market, only one call.

(3) A salesman always has a 25% chance of making a sale. For each call, examine the next number on the random number table. If the number is 25 or less, then a sale has been made; if it is 26 or more, no sale is made.

Advertising the Product

Product advertising in any quarter increases the salesmen's chances of amking a sale. It covers only the region or regions (I,II,III, and IV on the game board) that the company designates, and is effective in the current quarter only. Advertising costs $3000 per page, and a company may buy up to five pages of advertising in any region in any quarter.

Here are the umpire's instructions:

For each sales call within the region(s) in which the company has advertised, go to the next number in the random number table and determine whether or not there is a sale according to the probabilities in the following table. If the number is the same or below the probability percentage, a sale is made.

Pages	Amount	Probability of a sale
0	0	25%
1	$3,000	29
2	6,000	35
3	9,000	42
4	12,000	48
5	15,000	52

Research and Development

If a company can develop a superior product, it gains a competitive advantage. Usually, research and development have to be fairly continuous to achieve a product improvement, but a "crash program" may yield results in a relatively short time. The minimum research effort per quarter costs $10,000, but a company may invest more than that in multiples of $10,000.

The umpire notifies the company immediately when its research and development program has produced results, and all units scheduled for production in that quarter are considered to be equipped with the improvement. To find out the extent to which customers wil

prefer an improved product, $5,000 of market research (obtained from the umpires) is needed.

Of course, these ground rules can be altered to fit a company's situation more closely--just as the ground rules for other aspects of the Business Management Game can. A company manufacturing equipment for railroads may well want to use different units of research expenditure than would a company making dies for plastic products. The length of time necessary to get results from research also varies greatly from company to company, as does the cost of research to measure customer reactions to new products. These and other rules can--and in many cases should--be tailored to the realities of the industry.

The umpires will tell a company as soon as a competing team introduces an improved product in the market. The players can then counter with a stepped-up marketing effort or a crash research and development program.

If a company is interested in finding out the total industry research and development expenditures for the past year, such information is available from the umpires for $1,000.

In addition, the umpires should:

(1) Maintain a cumulative account of each company's expenses. After each break in continuity (a quarter without any R & D expenditures) and after each product improvement, start the accumulation over again.

(2) Make appropriate revisions of the probability of improvement. The cumulative dollar amount spent on research and development determines the <u>chances</u> a company has for obtaining a product innovation. Examine the random number table; if the next number is the same as or below the probability percentage, an improvement is achieved.

Cumulative amount	Probability of Improvement
$10,000	0%
20,000	0
30,000	0
40,000	2
50,000	4
60,000	7
70,000	11
80,000	15
90,000	18
100,000 and over	20

(3) Whenever a company achieves an improved product, increase all its sales probability percentages by 10. For example, if Company A has an improved product, this is the result:

	Probability of sale
Old product	25%
Improved product	+10
	35%

If Company A spends $6000 on advertising in one region and has an improved product, this is the result in that region:

	Probability of sale
Old product with two pages of advertising	35%
Improved product	+10
	45%

(4) As soon as all three companies have improved products on the market, cancel the premium of 10 for all three.

(5) If one company achieves two product improvements before one or both of its competitors have achieved any, increase all its sales probability percentages by 20.

Increasing Production

The initial plant which each company must build costs $150,000, and has a maximum throughput of 5 units each quarter. From then on a company may add other production lines for $30,000 each. But each such $30,000 increment will increase the maximum throughput by 5 .

A company must pay for increased capacity as soon as it decides to start construction. Construction time is nine months (three time periods), and only after completion may the first unit be put into "work in progress" for the new production line. The companies are not allowed to sell or otherwise dispose of excess capacity.

The total lead time in producing units in a company's plant is six months. First, production is scheduled, and this involves no financial outlay. Then in the next quarter units are put into "work in progress" and

must be paid for. In the subsequent quarter these units come off the production line , are added to inventory, and may be sold.

Total production cost contains a fixed cost and a variable element. The fixed cost is incurred each quarter, regardless of how many units are produced. At a maximum capacity of five units per quarter, the fixed cost is $6000, and the variable cost per unit is $3000. As capacity is increased by additional production lines, fixed costs rise and the variable cost per unit decreases. If a company, prior to adding a line, wants to know the exact costs it will incur at the next level of capacity, it can get that information from the umpires for $2000, but otherwise the umpires will inform the company what production costs are when the new line goes into production.

Units are added to inventory at actual cost. When a unit is sold, however, it is deducted from inventory at the average cost (total inventory investment divided by number of units in inventory).

The umpires should calculate the production costs at various capacity levels as follows:

Max. capacity	Total unit cost	Fixed cost per quarter	Variable cost per unit
5	$4,200	$6,000	$3,000
10	3,600	14,400	2,200
15	3,000	22,500	1,500
20	2,400	28,800	1,000
25	1,800	31,500	600

Financial Management

The management of a company's available capital is of critical importance. Each company starts with $400,000 capital and grow only through reinvested earnings. Profitability will be in direct relation to the skill with which the various parts of the business are kept in harmony with each other to achieve sound growth.

The price per unit of product is fixed at $10,000. When a sale is made, accounts receivable are increased by the total amount of the sale, and on the game board an accounts receivable symbol is placed on the fifth space in the "accounts receivable" column. Every quarter this symbol is moved up one space until after four quarters it reaches the top space and becomes cash. Competitive pressure in the industry forces the extension of credit; hence the one year collection lag.

If a company is short of cash, accounts receivable may be factored to get cash immediately. The

cost of doing this is 20% of the amount factored.

Appendix B

The following is an MSL model of parts of the game (for <u>one</u> "region") described in Appendix A--as seen from the point of view of a player wishing to investigate the game and see the effects of various strategies. It is presented here as an illustration of the use of MSL.

```
(*ACTIVITY HIRING
          (*PREREQUISTITES (*PRESENT (1000 CASH)))
          (*SCHEDULE ON CALL)
          (*PRIORITY 2)
          (*OUTPUT (SOME TRAINEE))
          (*TAKES 0)
)

(*ACTIVITY TRAINING
          (*PREREQUISITES
                    (AND (*PRESENT (1000 CASH))
                         (*PRESENT (SOME TRAINEE))))
          (*TAKES 3)
          (*OUTPUT (SOME SALESMAN))
)


(*ACTIVITY URBAN-CALL
          (*PREREQUISITES
                    (AND (*PRESENT (ASSIGNED
                                   (SOME SALESMAN)
                                   (SOME URBAN-CUSTOMER))
                         (*PRESENT (500 CASH))))
          (*TAKES .5)
)

(*ACTIVITY RURAL-CALL
          (*PREREQUISITES
                    (AND (*PRESENT (ASSIGNED
                                   (SOME SALESMAN)
```

```
                                        (SOME RURAL-CUSTOMER)))
                                    (*PRESENT (1000 CASH))))
        (*TAKES 1)
        )

        (*EVENT QUITTING
                  (*CONDITIONS QUITTING-PROBABILITY)
                  (*ACTIVITIES (SALES-CALL)
                               (*CANCEL)
                               (*REMOVE (THAT SALESMAN)))
                  (*ACTIVITIES (TRAINING)
                               (*CANCEL)
                               (*REMOVE (THAT TRAINEE)))
        )

        (*ACTIVITY ADVERTISING
                  (*PREREQUISITES (*PRESENT (3000 CASH)))
                  (*SCHEDULE ON CALL)
                  (*OUTPUT (1 PAGE-OF-ADVERTISING))
                  (*PRIORITY 3)
                  (*TAKES 1)
        )

        (*ACTIVITY R&D
                  (*PREREQUISITES (*PRESENT (10000 CASH)))
                  (*TAKES 0)
                  (*SCHEDULE ON CALL)
                  (*OUTPUT (10000 R&D))
        )


        (*EVENT PRODUCT-IMPROVEMENT
                  (*CONDITIONS P-I-PROBABILITY)
                  (*ACTIVITIES (R&D)
                               (*OUTPUT (1 PRODUCT-IMPROVEMENT)))
        )

        (*ACTIVITY PRODUCT-INITIATION
                  (*PREREQUISITES (*PRESENT
                                   (1 PRODUCTION-LINE)))
                  (*TAKES 1)
                  (*OUTPUT (5 UNITS-IN-PROGRESS))
        )

        (*ACTIVITY PRODUCTION-COMPLETION
                  (*PREREQUISITES (*PRESENT
                                   (5 UNITS-IN-PROGRESS)))
                  (*TAKES 1)
                  (*OUTPUT (5 UNITS))
```

```
)

(*ACTIVITY PRODUCTION-LINE-CONSTRUCTION
          (*PREREQUISITES (*PRESENT (30000 CASH)))
          (*TAKES 3)
          (*OUTPUT (1 PRODUCTION-LINE))
)

(*ACTIVITY FACTOR
          (*PREREQUISITES (*PRESENT (5000 A-R)))
          (*TAKES 0)
          (*OUTPUT (4900 CASH))
          (*SCHEDULE ON CALL)
)

(*EVENT SALE
          (*CONDITIONS SALES-PROBABILITY)
          (*ACTIVITIES (SALES-CALL)
                    (*OUTPUT (10000 A-R)))
)

(*FUNCTION SALES-PROBABILITY
          (*ARGUMENTS (PAGE-OF-ADVERTISING))
                      (PRODUCT-IMPROVEMENT))
          (*RETURN
                    (*SUM-UP
                    .25
                    (AD-FUNCTION
                      PAGE-OF-ADVERTISING)
                    (TIMES .10
                      PRODUCT-IMPROVEMENT)
          ))
)

(*FUNCTION AD-FUNCTION
          (*ARGUMENTS (PAGE-OF-ADVERTISING))
          (*RETURN
                    (*TABLE (PAGE-OF-ADVERTISING
                              *RESULT)
                     (0 0) (1 .04) (2 .10) (3 .17)
                     (4 .23) (5 .27)))
)

(*FUNCTION P-I PROBABILITY
          (*ARGUMENTS (R&D))
          (*RETURN (*TABLE (R&D *RESULT)
           ((LESSP R&D 40000) 0) (40000 .02)
           (50000 .04) (60000 .07) (70000 .11)
           (80000 .15) (90000 .18) (100000 .20)
```

This empty page was substituted for a blank page in the original document.