STNC
1090

Constructing and Analyzing Specifications of Real World Systems

by

Kaizhi Yuc

Department of Computer Science

Stanford University
Stanford, CA 94305

LELAND STANFORD JUNIOR UNIVERSITY
ORGANIZED 1891

rrMT.V

URGH, PENNSYL

CONSTRUCTING AND ANALYZING SPECIFICATIONS OF REAL WORLD SYSTEMS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Kaizhi Yue

September 1985

Abstract

The specification of real world systems is essential in developing application software and building AI expert systems. However, in formalizing our intuitions about systems, omissions and confusions cause errors. Correctly describing the behavior of a system can be a formidable task.

In this dissertation we propose that the semantic basis of representation languages lies in the mappings from observable situations to abstractions. In particular, we present such a mapping that serves as a framework for representing a broad class of real world systems and for analyzing and detecting errors in their specifications. At the basis of this framework are the causal and teleological interrelations that entail constraints among system categories. Specifications are analyzed against these constraints.

A language that incorporates these system categories has been designed. A symbolic simulator, which plays a similar role to an interpreter for a programming language, has been developed for this language. A comprehensive specification environment, DAO, has been implemented that can allow the user to edit, browse, and type-check as well as to simulate and analyze specifications.

Acknowledgement

It is impossible to list all the people who contributed to this research or to my computer science education during my years in Stanford. Let me remark that I am grateful for the friendly help I received from all of them, be it large or small. In the following, I will only mention those who directly helped me, one way or another, in this thesis research:

First, I would like to express my sincere thanks to Terry Winograd who, during the past three years of my dissertation research, has been supportive, understanding and sometimes tolerant of what I have been pursuing. As the principle advisor of this dissertation, he has been a patient listener and enlightening teacher. Very often I came to him with unorganized ideas and unjustified claims and left with a much clearer understanding of the subject matter. More importantly, he has contributed constantly to this research with his ideas, knowledge and judgments. In particular, his project on the specification language Aleph has had a profound influence on this research. Even on the issues with which he is critical and disagrees with me substantially, his broad and thorough knowledge and deep understanding of the issues helped me to pursue my own direction in a scientific manner.

I would also like to thank Terry for his help in preparing this thesis. He patiently went through many versions, corrected the mistakes in English and style, and made many suggestions for improving the presentation. The readability of this document has greatly improved due to the time and effort Terry has generously put in.

I cordially thank the other two readers on my reading committee. The influence of Mike Geneseretli on the direction on this research started earlier than the thesis itself. It was during the years of my association with the DART project that I realized the need for formal specifications when building models of real world systems. Through his sharp criticism, Mike helped me in identifying the true content of my research and in deciding how to present it. I am obliged to Danny Bobrow for kindly agreeing to be on my reading committee. Although skeptical of the objective of this research, Danny showed great patience in helping me to realize the importance of presentation and in its improvement. I am also thankful for his readiness to help me familiarize myself with Inlcrlisp-D and LOOPS, which arc the environments I built my system on top of. Lastly, it is through his sponsorship that I could use the computing facilities of Xerox PARC.

In addition, discussions with Ole Madsen, Martin Brooks, Weide Guo, and Chris Tong have been the source of encouragement to continue my line of research and very helpful in clarifying my own ideas. I should also like to thank Brad Hartfield and Susannah MacKaye who have helped in polishing the English in the manuscript and, as the earliest readers of the thesis, showed interest in the topics of the thesis.

Finally, I would like to thank Yuzhong, my wife. Without the support and patience from her, this thesis would not be possible.

This research has received support under a National Science Foundation grant (NSF MCS-81-04883) and a grant from System Development Foundation. The Xerox Palo Alto Research Center and the Center for the Study of Language and Information provided the computing facilities for this research.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Subject and Scope

This research concerns itself with the construction and analysis of specifications of real world systems.

By "real world systems," we mean systems that exist in the real world (or reality) rather than mathematical or other conceptual structures that exist by virtue of human cognition.

As to the question of whether the notion of *reality* is a fundamental conviction or just a convenient hypothesis, we will leave that to philosophers. For our purpose, the differences between them are either immaterial or have to be reduced to technical ones. We have found that real world systems exist and evolve in a frame of space and time, i.e., they exhibit observable behaviors. Furthermore, the behavior of one part of a system may affect the behavior of another part of the system, i.e., there exists something usually called "causation." It is these basic characteristics that make an investigation concerning them interesting.

It is important to note that although real world systems are necessarily physical, they are not necessarily physical systems as the term is used in the literature [Bobrow 1984]. In fact, our research has emphasized systems involving human activities, e.g., post offices, hospitals, and restaurants. These systems stand at a higher level of abstraction and physical laws are not appropriate for characterizing their behaviors.

By "specification," wo mean formal and precise descriptions stated at very high abstraction level. In addition, we will only discuss specifications in textual form. Very often, they are in some declarative form, for example, logic statements.

1.1.1 Specification as System Development Tool

Generally speaking, precise specifications are helpful in building systems. Actually, blueprints, engineering pictures and the likes all serve this purpose. A linguistic description is not as illustrative but it can express many things that graphic representation cannot.

A specification can be helpful in the following ways:

1. A specification is essentially a theory of the system being described. Writing a specification involves conceptualizing basic relevant properties of the system and expressing the constraints among them correctly. The work of writing a specification itself thus forces a process of clarifying one's intuitive ideas. The resulting specification can deepen other's understanding of the system as well.

2. A formal specification is precise. In this way, it serves as a good medium for communicating ideas about systems. For the same reason, it can serve as an agreement between system users and system builders.

3. A specification is abstract, i.e., it can be put at a very general level and omit lower level details and anything else that is irrelevant. For a system not yet built, it can thus serve as a high level design of a system, from which a prototype can be rapidly developed.

While the above is generally true of specifications of any system, in particular, the specification of a real world system can be useful in *software development* in two further ways.

First, it is directly related to knowledge acquisition in building model-based expert systems, since a specification essentially constitutes a system model.

Secondly it can help correctly define the interface for embedded application software because it helps people to better understand the behavior of the embedding real world system.

1.1.2 Validating the Correctness of a Specification

Errors in the Formalization Process: the Problem

Although formal specifications show great promise, they have not yet found wide use. The reason is simply because a specification, as a set of highly formal and abstract statements, is very difficult to write and easy to go wrong. This should not be surprising because producing a specification is the very first step in the formalization process. That is, before the initial specification, what we have is just intuitive ideas or natural language narrations about the system.

Furthermore, detecting errors is hard. This is because most current verification methodology deals only with verifying the consistency of a formal description or checking the correctness of one formal description against another formal description. However, a consistent description can still suffer from functional incompleteness and conceptual confusions. In the case of checking the correctness of one formal system against another, the correctness of the latter is taken for granted. But for an initial specification, its validation does not have the same formal grounds to stand on.

There are other reasons that make error detection hard. For example, the errors in a program could (but not necessarily always) be detected by running it. However, specifications are generally not runnable.

While, in general, we can attribute the reasons for making mistakes in producing a specification as human errors that any human activity is subject to, some specific sources of errors can be identified, which are related to the process of formalization.

In general, formal statements often have unexpected consequences to their human creators. That is because one's writing of a specification is often guided by intuitions shaped by the use of natural language, whose direct translation is often ambiguous and inconsistent. On the other hand, the formal specification resulting is to be interpreted mechanically by a computing device.

Specifically, in formalizing concepts, *multi-levels of abstractions and perspectives cause confusions*. On the other hand, in constraint formalization, *necessary context conditions may be neglected.*

For example, a description written in plain English of what typically goes on in a restaurant is seldom wrong as far as our common sense goes. However, its "encoding" into a formal specification can err in many ways. The cooking, serving, and eating of food can generally proceed in parallel but the events associated with a particular dish are strictly ordered. If the description represents the parallelism but fails to specify the order, we may describe a customer as eating something that has not yet been cooked.

It is also all too common for a specification to miss an event, reverse the order of events, or fail to mention a constraint among objects. All these are errors in constraint formalization.

Errors also occur when conceptualizing.

For example, one may define "dish" as a class of physically existing edible objects, which more or less corresponds to our intuition. If he then writes a specification that includes both "order(a customer, a dish)" and "cook(chef, the dish)" then something is wrong. This is because, first, the dish may not even exist when it is ordered. Secondly, if we think about it, the event ordering really involves the customer, the waiter he talks to, the sounds passed between them and, maybe, notes taken by the waiter. It does not involve the dish the customer eats. In fact, it may not involve any dish at all. What is involved is what is expressed in the sounds or notes, that is, the names of dishes, which are symbols for real dishes.

Here the trouble arises from confusing a symbol with its realization. Similar errors can arise whenever we deal with systems containing objects at different levels of abstraction (e.g., a file vs. the contents of the file, a name for a class vs. an object in the class).

Analysis Based on Domain Constraints: the Remedy

If we think about it, many of the errors mentioned above could be avoided or detected if we make use of our general knowledge about the domain.

A variety of things can be done. For example, we may have a distinction in our language constructs that distinguishes symbols from their realization. Or, we may require that only actual participants be mentioned in the specification of events. Or we may make use of the knowledge that no event happens to non-existing objects.

This general knowledge constitutes generalizations of knowledge about specific systems and concrete system entities. It is what we would call a *domain model.*

A particular kind of domain model we find useful and interesting is the "operational" model. By "operational," we mean that we are only interested in observable behaviors and their observable consequences. Moreover these behaviors are causally related. In this model, we categorize objects and events according to their inherent constraints.

Furthermore, we investigate a class of systems we call "stable operational systems." They proceed as sequences of discrete operations for achieving goals, and maintain a stable structure and ongoing course of operations. Widely differing systems, such as hospitals, microprocessors, ethernets, etc. can be given descriptions as stable operational systems, relative to a certain context.

The behavior of a stable system involves three distinct parts: an external environment, an internal structure (system per se), and some ephemeral elements, which come from the environment, play a role in the system, and then leave the system. In terms of the daily service of a restaurant, customers are in the environment, waiters and tables belong to "the system per se," and food and orders are ephemeral.

The internal structure is constituted of the components of a system. Because components have the property of permanence, being or not being a component will entail a set of constraints. Objects defined as non-components, for example, are subject to other constraints: they should be produced (or introduced), used, and destroyed (or made to disappear); in addition such events should occur in the right order.

The idea then is to incorporate the basic domain categories and the constraints among them into the constructs of a language. Consequently, a specification written in the language can be checked against these constraints.

1.1.3 Building an Integrated Specification System

The Role of a Domain Oriented Language

A basic tenet of this project is that to describe real world systems one should use a domain oriented language. A domain oriented language commits itself to a domain model by mapping categories in the language, e.g., class, isPart[1] etc., to semantic categories in the domain.

A formal language need not have semantic bases in domain categories to be used for describing systems in the domain. Predicate calculus is a perfect example. By not committing to a domain model, a language can be adapted to any domain models. But the user will have to build one for himself which is not an easy matter.

Some languages have domain models, but only partial ones. These include object-oriented languages, event-based system specification languages and AI knowledge representation languages, to name a few.

Object oriented languages have classes and objects which form a hierarchy or a more complex organization of knowledge about the domain. For example, we would typically declare, say, "person" to be a class and "student" to be its subclass. If a student John is to be represented in this scheme, he is often treated as an *instance* of the class "student." However, contrary to the user's intuition, the semantic basis for our being able to do this is not that the language has a model of the real world domain containing "person," "student" and John. The names of real world objects are only used in a metaphorical sense. The objects in object-oriented languages are really entities in computers. This is why the semantics for the language constructs are defined operationally, i.e., defined in terms of how the entities

[1] Although some object oriented languages, say LOOPS, have composite objects, they do not always require the user to declare some attribute as a part, e.g., the arms of a person. Thus this domain concept is not completely reflected in their language constructs. Moreover, this concept is only used in a metaphorical sense as will be explained below.

are handled in a computation process. E.g., how a property is inherited from superclasses is defined in terms of how a system subroutine is to look up the right property slot or "message" in the class lattice.

A step towards "domain-orientedness" is to explicitly introduce time and temporal relations."[2] Many behavior specification schemes have a concept of "event" (Although not necessarily the word) which reflects the temporal aspects of what happens in real world domains ([Peterson], [Lansky and Owicki], and [Lauer, Torrigiani, and Shields]). Being a domain category, the notion of time is an important cognitive dimension. Therefore, event-based specifications are relatively easier to read and write. However, the notion of event is more than just temporal sequencing. Moreover, introducing time and event is easier than introducing other domain concepts because these concepts are more amenable to mathematical abstraction and have a close metaphor in actual computation processes.

AI representation systems are intended to represent things in all aspects of real world domains. Some of these systems may make fairly explicit commitments to a particular domain model. For example, the notions of "defining attribute" and other semantic links such as "Has-part" in [Brachman] are clearly part of a domain model. However, quite a few AI representation systems try to stay at a level as general as predicate calculus. Although, by setting up a framework in a certain way, they do make some commitments, the commitments are often of a different kind: They commit to the control of reasoning and not necessarily to a domain model. For example, some languages allow a user to specify "default" values, or actions, or properties. The notion of default reflects the belief that there are certain states which real world objects are more likely to be in. However, operationally, this notion only means that, in the process of inference, if all other efforts fail, the thing designated as "default" is the last thing to try or the most reasonable thing to assume[3].

The specification system we are building has to commit to a domain model. Not just because we would like to use the constraints entailed from the model, as the last subsection suggested. It is also because the language is not a programming language. It is a language describing relations among real objects and the computations inside a machine can not possibly be used to define it. Besides this, we get well organized knowledge from the

[2]Most programming languages have an implicit time notion. Applicative and algebraic languages are timeless.

[3]Given the fact that AI languages are intended to represent real world domains, at least the ways an AI language is used (if not the language itself) implicitly commit to a model or are motivated by one. E.g., in events involving agents, the states of agents are often not changed before and after the events. That is, there is no observable consequences resulted on agents. However, these agents are specified in most specifications or descriptions of behaviors. The presence of these agents in specifications can find justification only from a domain model.

commitment to the domain model. As a result, we get economical storage and retrieval. This is in both a computational and a psychological sense, because domain concepts often naturally appeal to users.

Writing Analyzable and Understandable Specifications

We believe the key to analyzability and understandability of specifications is human cognitive dimensions and system inherent properties. And this is exactly the reason for adopting a domain model.

The design of the language and the specification style to go with it follow a simple philosophy. That is, get the description correct and understandable first, worry about its manipulation later. Besides the belief that it is very hard to get a specification correct, the idea behind this is also that many implementations are being developed to do the transformations for later stages of processing. As long as we have a well-defined semantics for our language it can be interfaced with other processing systems.

Many authors have discussed desiderata for system specifications ([Balzer and Goldman], [Holbaek-Hanssen Handlykken and Nygaard], and [Horning and Guttag]). One can hardly disagree with them. We only stress that one of the important goals in specifying systems is to accommodate multi-levels of abstractions and perspectives. A specification should give a picture of overall behavior, rather than reflect one simple view at one single level. For example, in specifying a packet routing system in a computer network we would like to see both how each packet gets treated and how each station transfers them.

As many recent authors have argued, a specification should be operational [Zave]. In terms of human understanding, operational specifications tend to have a sense of "causation" and are easier to comprehend.

All the above suggests that writing a specification should be a disciplined activity. A specification system should be built based on the assumption of educated users.

Building an Integrated Practical System

Writing specifications, like writing programs, involve intensive intellectual work. Like programs, specifications also go through life cycles. All this suggests that similar tools or environments should be built for specification systems. These tools should be combined into an integral whole. We have built such a system.

In particular we mention that our specification system is able to manage files containing specifications, to display specifications in various ways corresponding to various user perspectives, and to provide help when there are syntactic errors in specification. Most bookkeeping is automated.

The basic problem with the semantic processing of a specification is that is not "runnable" in program sense. However, various schemes can be set up to make inferences. In particular, we have developed a symbolic simulator that can process complex language constructs to infer behaviors of systems. This simulator uses a specially tailored theorem prover which is efficient for its particular data.

There are limitations in any checking scheme. The idea in building our system is to make use of constraints and heuristics to get as correct as possible. Since the language is not discerning enough to do everything automatically, the system can resolve ambiguities by querying the user, as necessary.

1.1.4 Semantic and Ontological Basis of Representation Languages

Another motivation for this research is to rationalize the process of description and theorize on those fundamental concepts such as identity and causality upon which we draw heavily in our practical work.

Unlike logic languages such as the predicate calculus, a representation language has an intended domain. The domain may be broad or narrow, but it is always some part of reality the language designer and user intend to model. Concerns for the domain have a strong impact on the semantics of the language.

This observation of ours, however, is often judged to be superficial. Since, after all, any formal language (representation languages are no exception) is supposed to be *interpreted* in some model in a model-theoretic sense. A theory (or description) in the language is correct, as long as all its consequences can find a match in the model.

In this way, representation languages are even somewhat awkward because the interpretation is possible only when we have given a translation in a logic language, say first order logic, to the language constructs. For this price, you get "syntactic sugar."

Here we would like to show that a formal representation language is more than its logic translation and the semantic import of a description is more than the logical consequences of its corresponding formal theory.

Problem of Interpretation

To illuminate the problem with the above view of interpretation, we point out that many things of concern to the user may not be represented as consequences in the formalism of a language, but rather have their roots in the underlying observations.

For example, the description of the events running and walking may both exist in the description of a large system, and, in terms of the description, they may both have the same consequences, i.e., moving the agent some distance. By "logical consequences," in general, we mean things that can be logically inferred from a statement. In this particular example, if for both "run(a person, position A, position B)" and "walk(a person, position A, position B)" one can infer "at(the person, position B)" and can only infer this, then the logical consequences of the two event descriptions are the same. Of course, a quantitative description may be able to put more in the description of the final states, i.e., consequences, of the two types of events. For example, it may also have the actual time when the event is finished. Very often, however, we will be dealing with a qualitative description. In that case, the difference in durations may simply be ignored. Still, in the practice of most people (as is evident from the specifications we examined), when specifying a particular process, say, delivering newspapers, one may prefer, say, running instead of walking as the description of part of the process. It seems that in spite of the fact that the logical consequences are the same for the two events, one does not want to use them interchangeably (e.g., categorizing both as "walk"), nor one would like to use a generalization of both (e.g., categorizing both as "move").

Our first question is what is the reason or rationale for this preference in writing descriptions. The answer is simple: One would like to make the distinction one has observed. If one has seen someone doing something running, one would like to be specific and say that someone is *running* rather than use the vague term "moving."

Then, of course, the second question is how we can make this distinction if the consequences of running and walking are specified to be the same, since, as we all know, a formal system does not derive any semantics from the suggestive English symbols (in this case from the words "run" and "walk"). Interestingly, the answer lies in the simple fact that we have used two distinct symbols which denote two distinct formal entities, and the two entities in turn, according to a reasonable abstraction procedure, are the mapping results of two distinct kinds of observations.

In another words, we now have a case where the logical consequences for two events are specified to be the same but they are still taken to be distinct because of the distinct underlying observations. The consequences of the distinctions between the two kinds of events may get lost when one tries to formalize or generalize, but they will remain distinct to the observer, nonetheless.

One might argue that since two distinct symbols are used, then the formal descriptions for the two kinds of events are distinct, and there is, therefore, nothing wrong with the formal specification. We do not dispute this. What we would like to point out is simply that the distinctions intended in using the two symbols cannot find their justification in the formal description itself. The justification has to resort to a more refined model of the world one is describing.

Theoretically the problem comes from how we view interpretation.

In logic, the standard notion of interpretation is no more than assigning non-grammar symbols to entities in a model. Here, by "non-grammar symbol", we mean those that are not contained in the notation of a logic (e.g. "A " or "V", or simly "(") but are the symbols standing for entities in the real world.) The logical consequences of the formal theory should be confirmed by the relations between entities in the model. As to where the model is from, the logic does not care.

If the language is intended to describe a domain, the situations differ. For example, if I use the string "msg" to mean a kind of entity, "message," in my formal model, I should be ready to explain how one's observations can be mapped into this kind of entity, i.e., "message." If, in my language, no "abstract object" is allowed, I cannot use "pass the message" to mean what we usually mean when saying that. This is because, if the message is in the form of physical sound, then "the message" has disappeared when I have received it. I have to "speak" and my words have "content equal to the content of the message."[4] As we tend to talk about things at many levels of abstraction, mapping rules of this kind are necessary for the efficiency and even possibility of communication. This is because what makes someone else's description alien to me is not always because it describes things completely new to me, but often because of the way it is organized. When the mapping rules are explicit, it is more likely that misunderstandings can be avoided.

The Need **for** a Semantic Basis **of** Specifications

The problem has not been seriously felt for a simple reason: The linguistic and cultural background have been assumed implicitly as the mapping procedures.

This problem, however, becomes evident in system specification languages.

The research in system specification languages has been concerned itself with modeling operations in real world systems, e.g., hospitals, post offices, etc. [Winograd 1984] [Guttag et al.]. It should be obvious that in specifying a system or situation what concerns us is no longer problem solving strategies (which have been the focus of problem solving languages)

[4] "Content[11] here can be taken as a function.

but the correctness and understandability of the descriptions, per se.

As many people have experienced, the writing of specifications is as error prone as the writing of programs, if not more so [Horning and Guttag]. The problem is often worse, because a specification can not be tested by running it.

The error often arises in formalizing our intuitions. As we pointed out earlier, one should be aware the distinctions between a symbol for a dish and a real dish. If this distinction is not made clear we get into trouble.

There is also a positive side of making distinctions. For example, if a patient comes to **a** hospital we expect him to leave. In a restaurant the patron comes and goes and the dishes are cooked and eaten. There seem to be classes of objects in any system that appear or get created and then disappear or get destroyed. And it is certainly true that they appear or get created first. By making distinctions and generalizing on them, we will be able to find some general constraints to check the correctness of specifications.

The Relation between Semantics and Ontology

Because the terms "semantics" and "ontology" (and, respectively, the adjectives "semantic" and "ontologicar) will be used frequently in this report, we will briefly discuss their connotations and the interrelation between the two. We note however that their uses do not entail any definitional power. They function just like a comment to help the reader to get the technical content.[5]

Usually semantic statements are statements regarding the nature of our description framework. "Mammal is animal" is a typical semantic statement.

An ontological (or metaphysical) statement is supposedly made upon some fact independent of our descriptive apparatus. For example, "physically speaking, a part is smaller than the whole" would have to be true[6] whatever the media to communicate it can be.

Purely semantic or ontological statements are rare if they exist at all. But this does not mean that one can not draw a boundary between the semantic and ontological *aspects* of a statement.

In practice, if in the context of a discourse the objects and events referred to by a statement can be considered to have well known denotations then the statement must be communicating about the state of affairs, i.e., ontological. On the other hand, if the purpose of the statement is to clarify the connotations themselves, e.g., how we classify, how we

[5]The use of word "ontology*[1] is controversial in philosophy. I would have used "metaphysics" instead if not for the wide use of the word in AI literature.

[6]One may raise question about its truth, but let us assume for the moment that there is no controversy.

assign names, and how the references of words relate, it should be considered semantic.

In theory, there is a thesis that "since any statement about reality has to be put into some language it is therefore semantic in nature." This thesis has ignored the fact that various descriptions are used for the same reality that is independent of our perception or cognition, although its description is not. And this independence will enter any statement we can possibly make about reality.

Of course, the thesis has partial truth (a la Hegel): Any statement has a semantic aspect. In this way, semantics "affects" ontology. We point out that there is influence the other way around. In particular, although it seems one can pick facts randomly in putting together a description of some sort, the most commonly selected facts are causally related. As a matter of fact, this is the intuition behind the definition of the completeness of specifications in our theory.

A Methodological Remark

Lastly, a few words about the general approach of this research.

In this research, we try to balance the theoretical side and pragmatic side of the problem. In approaching a problem, we always attempt to give it a precise formulation as we believe this is the way of guaranteeing the generality and correctness of the work. However, we do not attempt the kind of mathematical rigor logicians find mandatory.

If we draw an analogy, a physicist may calculate derivatives of a series without proving first that the series is convergent. This is, of course, not mathematically rigorous or even correct. But the reason they do it is that from the physical meaning of the formulation the series "has to be" convergent. Similarly, in our case, we will not engage ourselves in mathematical details if we find it not essential in illuminating the domain concepts but just a complex manipulation of complex formulas.

Another characteristic of this work is that we put strong emphasis on the conceptual aspects of system descriptions instead of the "procedural" or "temporal" aspect. This is based on the feeling that works stressing the latter aspect, e.g., Petri-net, CSP, etc. are well known and well accepted. But conceptual understanding of systems is still lacking and little has been worked out.

1.2 Overview of The Thesis

The thesis is divided into two parts.

The first part, "Formalizing the Intended Domain of a Representation Language" deals with theoretical issues which have emerged in this research and serves as a theoretical foundation for what the actual specification system, DAO, does. For someone not theoretically oriented this part can be skipped at the first reading but some specific sections (indicated below) should be read later. Otherwise, many things we do in the actual program will not make sense.

This part starts from presenting a naive model of the description process then modifies it into a formalized model, which lends itself to mathematical manipulations. The key idea is model mapping. Some interesting mapping properties are discussed. The second half of this chapter (chapter 2) is a bit too theoretical for "application-minded" readers. It can be skipped on the first reading or even ignored altogether.

The following chapters deal with all the basics in this research. They begin with a very easy chapter on basic categories that talks in vague terms but render intuitions. Right after that, we will get into some difficult material, i.e., the representation of time. A reader need at least to scan it to find out what symbols are defined there and get a rough idea of how all that works together. For a reader serious about temporal representation this chapter raises some deep issues to ponder.

A separate chapter is dedicated to a discussion of causation, just to make sure we have a rich enough language to characterize the domain categories.

The formalized categorizations of events and objects then will follow.

The chapter on models of systems is another important chapter. Here we analyze the systems we observe in everyday life and sort out important constraints. If the reader does not like logic formulas, he should at least read the English text.

The second part of the thesis explains how the actual program, DAO, works.

This part starts with a chapter on the DAO specification language.

We spend one chapter (chapter 10) on the system's functions as a whole. The section "a scenario" gives a quite detailed step by step illustration of how one can use the system to generate and check specifications. The second section presents the working environment. It lists what the system can do to make your life easier but has nothing conceptual. The last section, "experience with DAO" discusses the author's experience as a user with DAO. You may want to postpone reading it until the end.

The chapter on internal representation is not too helpful for later reading. This is because the presentation of simulation and analysis has been kept on the algorithm level. However, this chapter discusses various trade-offs in selecting representation systems, theorem provers or even particular knowledge base management routines. It may be of interest to serious AI system designers.

The next two chapters present general algorithms for the syntactic and semantic processing of specifications written in DAO.

Separate chapters discuss the related works in both the areas of formalizing real world domains and constructing and analyzing specifications.

1.3 Major Contributions of This Research

In the theoretical aspect[7], we have established a detailed model of description process based on model theory. In this model, we formalized the process of what is commonly called "abstraction" into the notion of mapping between observational models and abstractional models. In so doing, many interesting properties of the abstraction process can be investigated with available powerful mathematical tools and within well established mathematical frameworks. For example, we talk about the soundness and completeness of the mapping as we would with deduction systems. Furthermore, our analysis reveals that the folklore about abstraction being "ommitting details" misses the point. The essence of the abstraction process is selecting *relevant* facts. One of the potential criterion of relevance is causal relation.

This formalized model of description not only is an elegant and pertinent rational reconstruction in its own right, but also makes all ensuing axiomatizations of domain concepts well-founded.

The second significant work is the operationalized (or operational) domain model. This work is unusual because we try to use behavior alone to categorize human activities and their consequences. For this purpose, we introduce concepts such as information object and social relation. Our definition of events is more enriched than that offered by many other theories as it identifies an event by more than state changes in ordinary sense. By introducing the notion of interaction and axiomatizing causal relations, we can ask more questions about an event than its direct logical consequences. The axiomatizing of causality, interaction and other relations have their independent value as well.

Thirdly, the notion of a stable operational system, sketched in the final chapter of Part I, as a scientific abstraction like frictionless surface, is a powerful idea. Because of its wide practical applicability, it can be very useful in understanding the structure of systems.

[7]In general, one is not the best judge of what one did. This is in both a spatial and a temporal sense. That is, the value of research works should and will be tested by time. Therefore the title of this section is really a misnomer. What I would present here is just a list of things that I think to be novel and significant.

On the other hand, it is another cornerstone of our intuitively feasible checking scheme. Even the simple and "obvious" notion of "creation of an object" is really based on it. This is because only when there exists a boundary between a system and its surroundings, does the creation (or introduction) of an object have definite meaning. In fact, only a particular class of objects in stable systems, namely noncomponents, go across this boundary and get created.

The temporal representation scheme invented for system description is not only helpful in producing concise and understandable specifications, but also a coherent and novel approach for representing time and time-dependent facts. This scheme allows us to treat relations and events uniformly as predicates, which makes it easier to express a broad class of facts concerning both relations and events.

I am reluctant to call it a temporal logic because I did not do the proof work that is generally required for introducing a logic. But, still, being (mostly) defined in terms of first order logic, this representation gives all the sentences written in this scheme a precise meaning.

In the pragmatic aspect, we have designed a language that incorporates the domain categories (which in turn bear with them domain constraints). This language can be used by educated users to produce specifications that are checkable in many ways.

We have introduced the notion of simulation of specifications. In this simulation, axioms as user-defined constraints are used to check the validity of specifications. Our simulator can handle cases like sets of objects and concurrency, both of which add considerable complexity to the processing algorithm.

The notion of analyzing simulation traces is based on the ideas of domain constraints. In this sense, we do not see it as a new thing. However, it is some novel semantic processing device as a practical system.

Overall, we have built a comprehensive specification system that allows a user to create, edit, type check, as well as simulate and analyze, a specification. On this system, we have done some experiments that have helped to refine and test our above ideas. In building or experimenting with this system, we have done series of works to put together all the ideas. Although we do not think any one of them taken individually is a significant contribution to human knowledge, taken as a whole they are a well-organized working system and represent a successful experiment in specification research.

1.4 Formal Notations Used in The Thesis

We will base our scheme on a first order logic with equality. The notation we use for ordinary logic symbols mostly follows their use in [Manna]. This includes

A , V , -., s, V, 3, and t=.

Because we use set inclusion and logic implication equally frequently, we will use "D" for set inclusion and "-»" for implication.

Besides, we will use following notations for common place predicate:

≥ as greater than or equal,

≤ as less than or equal,

•*fi* as not equal.

We use two abbreviated forms in quantifications: a quantified varaible can be followed by a sort or a set. They are defined by the equivalences below.

Vx:X [p(x)] ≡ Vx [X(x)->p(x)]

and

3x:X [p(x)] ≡ 3x [X(x)A p(x)]

are for sort variables.

Vxex [p(x)] ≡ Vx [[xex]-»p(x)]

and

3x€X [p(x)] ≡ 3x [[x6X]A p(x)]

are for set variables.

We have following convention in the use of predicates that are special to our theory.

Some predicates are defined explicitly by a set of axioms. Their uses are standard. There are many concepts that are characterized by a set of axioms but the concept itself is never defined as a predicate. For example, this is the case with "physical object," "physical event" and so on. We will use the names of the concepts as predicates directly for convenience. Sometimes we can not give a formal characterization of a concept but just explanations, the name of the concept can still be used as a predicate. We take it as a primitive. For example, "Agent" and "object" will be so used.

For the convenience of the user, a list of the conventions of symbol uses is attached as an appendix. Also, we prepared a subject index. This index will show the places where the indexed items are defined or explained.

PART I

Formalizing The Intended Domain of A Representation Language

Chapter 2

A Model of the Description Process

2*1 A Psychological Model of the Description Process

2*1.1 Rationalizing the Process of Formal Descriptions

We have learned from the psychology and linguistics literature that a description is arrived at through several cognitive stages. First, one looks at the world and sees something ; this is called "sensation" and "perception." For example, coming to San Francisco, a tourist may board a cable car. He sees how the car is driven, how the conductor collects the fare and feels how thrilling it is to ride. Then he tries to put these things together into an informal account of what a cable car ride is all about. This informal account we call an *informal mental model.* If he is observant and thinks clearly, we expect he will tell a *good* story in his postcard to his friend. Telling *the story,* i.e., the selection of language (French, German etc.) and the specific words and sentences of that language, constitutes the last stage of this description process.

When we say a story is "good," we do not only mean that the composition is written grammatically and with correct vocabulary. More importantly, we want it to be coherent, more or less complete, and not contradictory. In short, just what we would expect from a good story teller.

Producing a formal description is very much like telling a story. The difference is that one uses formal languages instead of natural languages such as French or English. One has to first transform one's informal model to a formal one, which can then have direct correspondence to the symbols in the formal language.

An illustration of this conceptualization of the description process is shown in Fig. 2.1.

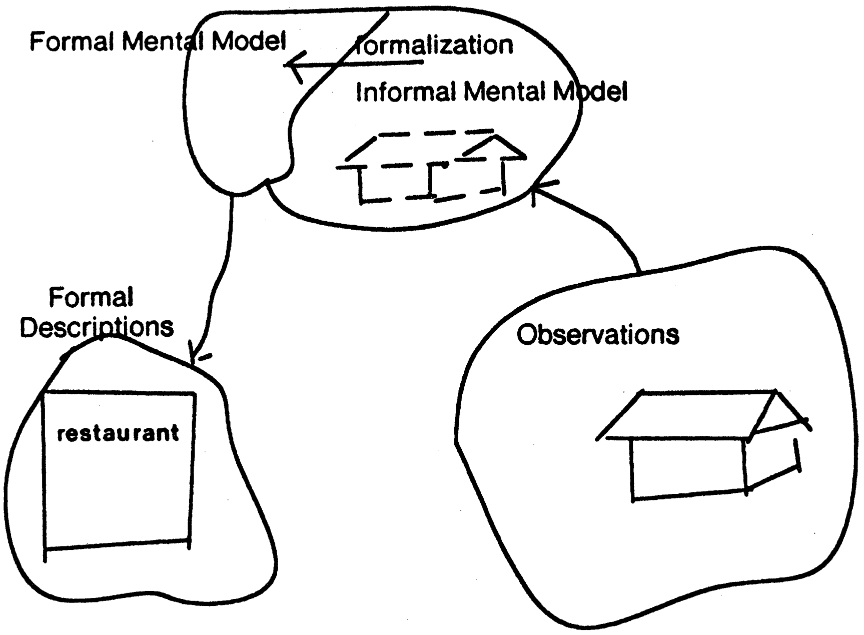


Figure 2.1: A Psychological Model of the Description Process

Let us take a closer look at what is in this picture.

The observation provides us with "raw materials" for selection and organization. As sensory data, these are fragments or small pieces of information that capture individual facts in the world. At this level, one sees trees but not the forest. On the other hand, this part of the description process is the least disputable. Usually if one sees a chair, another will not report observing a TV set. Therefore, this level often provides a basis for people to resolve their differences at higher levels[1].

Another aspect of this stage is that it is at a high enough level to allow the use of words to physically identify individual objects and relations[2].

This characterization of observation more or less coincides with the notion of "basic categories"[3] developed in cognitive psychology [Rosch]. Experiments have showed that people tend to perceive and remember objects at a certain level. At that level, the objects in a category have common functional relations to people's activities, show common patterns

[1]Observers may differ on what they see in finer details when they have different expertise in the subject matter. E.g., a furniture salesman may observe much more than a layman. But this is another matter.

[2]Here "individual" is used in the sense of "uniformities across situations" as suggested in [Barwise and Perry].

[3]The reader should be forewarned that the use of the terms "basic" and "categories" here differs drastically from our use in the forthcoming theory.

in motor movement relating to them and have common perceptual appearance. The words standing for these categories are most frequently used. For example, most people would be more likely to refer to something as "dog" rather than as "animal" or "German shepherd."

Next, we see a cloud called "informal model" in the picture. Here we use the word "model" in a very loose way. It means more or less organized knowledge of something. If it is about cable cars, it may be the routing patterns known to a car operator or the first impressions of a visiting tourist. In either case, it is the result of assimilating the materials acquired in the observation stage.

This assimilation or organization often entails what are known as abstractions such as generalizations, aggregations, and classifications. E.g., one may decide to call both the writings on a paper and a sequence of sounds, "messages." Or one may decide not to talk about the transistors and capacitors but only "the radio."

There are dozens of theories of this assimilation process which are psychological in nature. If the tourist follows Minsky's advice, he may be filling his slots in "city transportation" or "tourist attraction" frames [Minsky][4]. If he believes in Conceptual Dependency [Schank and Abelson], he may be checking the scripts of the same kind. If our poor tourist has never been to a city, then he might be moving up and down version spaces [Mitchell] to learn the commonality between a cable car and a truck and the difference between a conductor and a driver.

In any case, from this informal model, one is ready to tell others about something in an organized and selective way. As a matter of fact, the knowledge in this model is often shaped by the language used. We also should point out that this level is really many levels in terms of abstractions. One can talk about riding a cable car as a ride, a tourist experience or part of a vacation.

In the informal model, the context and background knowledge are preserved implicitly, partially because of the use of natural language. In the formal model, the next higher cloud in the figure, they are no longer available. All the relations among primitive entities, i.e., the knowledge higher up than primitives, are to be made explicit in the formal model. If something is not mentioned then it is lost.

However, the meanings of the primitive entities are supposed to be found in observations. Here, the construction of the formal model is even more influenced by the language used, because formal languages are seldom as versatile as natural ones. This influence, we note, is not so much because of syntax or notation but more from how the languages "divide up" the world. (cf. the section on domain-oriented languages in Chapter 1.)

[4]But Minsky's frame idea is also intended for perception, e.g., vision.

A real example of going from an informal model to a formal one is the process of knowledge acquisition in building knowledge based systems. Much work is being done in this area but it is still the bottleneck in system building.

Being concerned with producing "good" descriptions, we notice that there are some basics in constructing a mental model, be it informal or formal. For example, if one confuses things or treats the same thing as two different ones, or if one narrates facts in random order, the description will likely be confusing or incorrect. On the other hand, an appropriate way of organizing facts makes a description more understandable and less fallible. So we are tempted to seek general principles that can guide our description activity.

However, our current characterization of description process, although faithful to reality, is too much of a psychological model and thus does not lend itself to a simple formal treatment. We conjecture that there can be a collection of simple rules for building a formal model. But when you take into account all the idiosyncrasies and variations in observations and informal models, they are no longer simple. And this is often the grounds for the argument that "there can not be a formal treatment of language design."

The measure we take is to introduce an ideal model of description.

2.2 The Ideal Model of Description

To make our observations amenable to formal treatment, we put them into a formal model. We approximate our informal and idiosyncratic ideas about reality [Labov] to a formal model, which we call the "observational model." ("observation" for short where no confusion will arise.)

This model corresponds to our "true observation." This model is basic in the sense that all entities in it are atomic or primitive. Any organization or abstraction happens at a later stage. (In the future, the term "primitive" will also be used for entities at more abstract levels. But "atomic" is reserved for this basic level[5].) For example, we may put a set of relations and events such as "connected(Terminal-A, Terminal-B)", "HighPowerLevel(Terminal-C)", and "increment(Voltage-of-Terminal-D)" and so on in a formal structure, e.g., formulas in naive set theory. They represent all the relevant observations we obtain from observing the behavior of a radio set at the level of electronics.

This hypothethized model is formal in that we do not try to relate it to any psychological

[5] We recall that informal models are often higher level conceptualizations or generalizations. A part of the informal model is "permitted" in the observational model, if it can be considered "atomic" for the problem at hand. This means one can not only "observe" (in this ideal model) trees but also forests. But since a tree would be a part of a forest, one can not make forests and the trees in the forests atomic at the same time.

substrates. However, we note that our intuition in that respect is the rationale for the introduction of this model. Also, many properties of our observations mentioned above, such as being fairly uniform among individuals, make this hypothesis a justifiable one.

This model is general in that we assume all observers will produce the same model for the same course of events. Idiosyncrasies disappear. Being a generally accepted model, the observations are more likely to be correct, and this gives another important property to this model: It is consistent. Note that in an individual's informal mental models, inconsistency is often present.

Relative to the observational model, the formalization of the mental model at a higher level of abstraction is called an *abstractional model*, or abstraction for short. It can be viewed as a result of mapping from the observational model. For example, in the case of the radio set, the relations involving the terminals, the wires, even the transistors will be replaced by entities such as "high frequency amplifier", "audio frequency amplifier", "speakers" and so on, or even simpler, by entities such as the mechanical "control unit" (including the knobs and buttons), "electronic unit", and "sound unit". Through this mapping, the identities of primitive objects and events are changed. We note that both models are formal structures. We can use all available formal methods to investigate relations between them.

Now, the process of producing formal descriptions can be viewed as consisting of the following stages: from reality to observation through perception, from observation to abstraction through a functional mapping, and from abstraction to description through an entity-symbol assignment (see Fig. 2.2). The inverse of description (in this sense) is the process of *interpretation.*

Of these three stages, the first stage has been explained clearly enough, and the last stage, assignment, is the inverse of what is traditionally viewed as interpretation. It assigns each non-grammar symbol to an entity in the abstractional model. The reader is reminded that non-grammar symbols are those that are not contained in the language's concrete syntax but stand for real world entities.

The mapping stage is our major interest here and will be discussed in the next section.

2.3 The Mapping from Observations to Abstractions

The mapping for a description scheme should answer questions such as: "What is an entity of a certain kind in this model" or "what is the way in which the entities are composed." A good answer to these questions would be procedures that help to produce correct specifications.

There are very general rules for mappings. These include requirements that the same

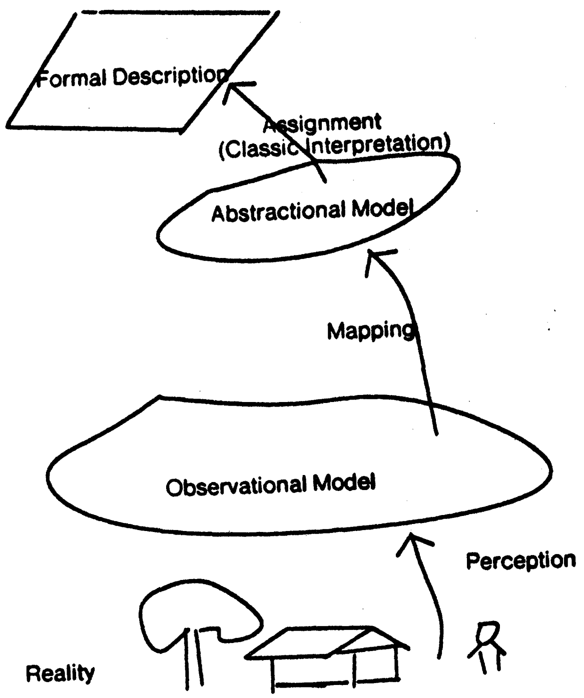


Figure 2.2: An Idealized Model of the Description Process

phenomena (whatever "same" means) observed at different times be abstracted into equivalent forms. Similarly, a scene viewed within a smaller scope or within a larger one should result in equivalent abstractions.

There are also relatively specific rules for mappings. We will provide two examples illustrating the points. They are related to the following intuitions: First, we do not want to see repeated mentioning of the same atomic event in two primitives in the abstractional model. Second, we do not want an atomic event in the observation to be counted in one scene but ignored for another scene.

Assume a concurrent event having E_1 and E_2, we ask what would happen if we allow them to share some component events as depicted in figure 2.3.

To see the consequences of this "component sharing," we will visit a fictional restaurant which is built from a design allowing the above practice.

Imagine that I come into the restaurant: When I give my order to a waiter, I get a coupon to exchange for my food. I then order from another waiter and discover that I get the same coupon for the same food. I then pay for one coupon and get a receipt. I show this receipt to the first waiter and get food served. After finishing eating it, I show the same receipt to the second waiter, and the waiter serves me a second time. So I get to eat

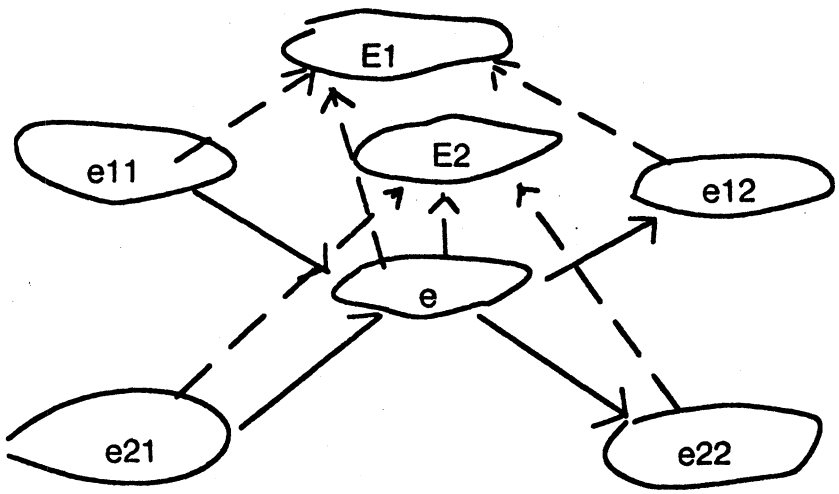


Figure 2.3: Sharing an Event in a Mapping

twice while only pay once.

The way the restaurant is run is weird. However, if we interpret the component events in the concurrent event of E_1 and E_2 in figure 2.3 as the two dining events where e11 and e21 are "ordering" and e12 and e22 are food serving, then the above story would be an concretization of the specification.

One may argue that it is this particular specification, not the general way of practice of "sharing component events" that is to blame. One may suggest, say, "Suppose whenever I describe two concurrent orderings and eatings I explicitly mention two payings."

First, we note that the matter can not be corrected by adding a particular component in a particular event. As long as we allow the sharing, they may appear at lower levels recursively. E.g., if paying involves going to the cashier, showing the coupon, passing the money and going back, I can have the part of money changing hands shared and still accomplish two paying actions. Moreover, if one does not want to further decompose an event(here E_1 or E_2), this means he just can not describe the events at all.

If we find that in most cases this procedure of selecting facts will produce an error, we want to make a provision requiring that no event be counted more than once, which then becomes a principle of a description scheme or part of the definition of a language. Similarly there should be provisions for the mapping of objects. In the case of a abstractional model of our radio set, we do not want the same device, e.g., a transistor be counted as the parts of two modules[6].

[6]In some cases, we do see the "sharing of component events or objects". This happens when the architecture of the system is not modular and the same component events or objects play roles in different

As a second example, an observation often has things that are not mapped to its corresponding abstraction. There should be procedures for allowing omissions of certain facts. For example, comparing a restaurant with its specification, if, besides the specified food servings, we see another episode of customer ordering food and eating it, we would claim that this restaurant does not confirm to the specification. However, if there is only one tea serving event specified, we may simply ignore the second or third tea servings when we observed it, provided the main goals of the restaurant operations are taken as customer eating and paying. Intuitively, the criterion for the selection of observations is whether an event has a causal relation with our particular interest. Namely, an observed event should be reported if it affects a reported system goal. Again for the example of our radio set, we observe that if an object or event is directly or indirectly related to the sound output of the radio, then it should be included in the abstractional model. E.g., the color of the knob is usually not necessary to be mapped (even if it is observed) as far as the electronic functioning of the radio is concerned. On the other hand, the wire that connects the speaker and the audio amplifier has to be present in the abstractional model.

2.4 Formal Properties of the Mapping

2.4.1 Situations, Domains and Their Counterparts in Logic

We now formalize our intuitions. We will base our formalization on a well-established theory of models of reality. This is the theory of situations developed by Barwise and Perry in their work on situation semantics [Barwise and Perry]. Their theory has been developed for the semantics of natural languages but we are interested in the semantic bases of artificial languages. Consequently, many of their specific definitions are not necessarily useful for our purpose. However, since both works are concerned with the modeling of reality, we find their basic framework can be adapted to our need. In the following, we will define a notion of situation that is essentially the same as theirs. The rest of our own framework, such as model mapping, will be introduced on the basis of this fundamental notion.

We define a *situation* as a set of relations or events with time, locations and truth indicators T or F. The truth indicators T and F stand for Truth and Falsity respectively, meaning whether the event or relation is observed or not.

A relation or event can be written as

module units. We note, however, that although occasional compromise may be possible, in general, in such cases, these module units should be viewed as a single unit, because one can not predict how and when the consequences would be different if the different roles are played by objects or events with different identities.

p(x,y,...t,tM, T) **(2.1)**

or

p(x,y,...t,t',l, F). **(2.2)**

In the latter case, we do not observe the relation or event p for the time between t and t' at location 1.

A situation Situ is written as

Situ = {Pi(x_t-,y_{tr}. .a^btv-.t^'^bool,·) |i}

where p^ is a relation or event, x_t-,y_{ir}.. are individuals, aj,b_tv-.are non-individual parameters[7], t,- and V are respectively the time points when p begins or ends taking the truth value $bool_t$, l_i is the location and bool» is the truth value. A situation is empty if the set is empty. For example, the set of relations and events involving the parts of a radio set is a situation as defined above.

There are special kinds of situations that are of special interest to us. These are situations containing "everything we care to talk about or more." They are called *domains.* The foregoing discussion on observations and abstractions provides intuitions for them. Being mathematical structures, they are *models of the world* in model theoretic sense. For example, if radio sets are all that we can observe or care to, then the situations involving them would also be a model of the world. However, usually, our domain is broader. E.g., they may be circuits or physical systems.

In the future, if a situation is a singleton, we may write its only element instead. E.g., we will write p instead of {p}where there is no danger of confusion.

For each relation or event in our domain, we can have a literal without variables, such as

p(x,y_r..t,t\l) **(2.3)**

or

-p(x,y,...t,t\l) **(2.4)**

as its logic counterpart. In the future, we will have the convention that an individual entity in the model corresponds to its counterpart in logic formulas just in the way (2.3) is for (2.1) and (2.4) for (2.2).

Corresponding to a situation, i.e., a set of relations or events, we will have a set of logical formulas formed by taking the logical counterparts of the elements in the situation. In the future, we will subscript a symbol standing for a situation with "fset" to mean that it has changed to stand for the corresponding formula set. For example, we will write $Situ_{fset}$.

[7]Non-individual parameters are also called "pure values." Their ontological status is "abstract object" which will be discussed at length in future chapters.

Situ/$_{ae}$*t* itself corresponds to a logic formula which is the conjunction of formulas in the formula set. This will be denoted by Situ/$_{con}$y$_{unc}$t»on* But we are more interested in the deductive closure of this set of formulas, which will be denoted by Situ$_{c}$/$_{O}$sure« We note Situ$_{c}$/$_{Odur}$e is a theory in the sense of model theory. For example, from the model of the radio, we can obtain a corresponding theory of radio sets.

A theory such as Situ$_{c}$/$_{oaurc}$ is not equivalent to its corresponding set of formulas in the sense that its logical consequences may not be in the set of clauses of Situ/^^. In particular, we note all the formulas[8] with logic symbols such as "V ," "V," or "3" would be in the consequences but not in the original formula set. For example, if in the domain we observed that all people in the domain, e.g., John, Mary, and Joe are all students then we will have formulas like "student(John)," "student(Mary)," and "student(Joe)." Then we can have a quantified formula

Vx[person(x)—^student(x)].

This formula is not originally there but it is in the theory by deduction.

The corresponding formula of an empty situation is equivalent to T if it is being taken as conjunctive with a relation or event, it is equivalent to F if it is being taken as disjunctive with a relation or event. That is, for a formula p,

if Situ=0 then [[Situ$_{c}$/$_{oaurc}$A p]=p] A [[Situ$_{cJO5ure}$V p]=p]

The apparent similarity between a situation in the model and a formula in the theory in terms of representation may give rise to a question: Why do we need to talk about this correspondence, why don't we just work "inside the model," or why don't we go directly to the theory?

We can not stay in the model because there is no powerful deduction mechanism available for us to reason about things we observe. On the other hand, going directly to theory, i.e., setting up formulas without asking their ontological importations, will often yield conceptual confusions.

2.4.2 Mapping of Models

A model mapping MAP is a function from one domain to another domain or from one model to another model. In the context of our research, only two domains concern us: the observational model and the model resulting from this mapping, which we call the abstractional model, or abstraction:

[8]The term formula can refer to both atomic formulas and logic expressions formed by connectives. If we need to be specific, we will use "clause," "literal," or "atomic formula" instead. Otherwise, if we only care to differentiate between model and theory, we will simply use "formula."

MAP: Observation —•Abstraction

In the future, we will also refer to a mapping as "abstraction procedure" or "abstraction rule" or sometimes as an "equation."

A mapping can be very general. For example, we may specify when a set of objects in the observational model can be viewed as a single abstract object after a mapping. Mappings like this determine how a mapping "divides up" the world. This will be elaborated in the first part of the thesis.

A mapping can be specific. Specific mappings define particular description primitives or semantic bases [Winograd 1984] for the abstractional model. For example, in the case of our radio set, we may have the network of connections among transistors, resistors, and capacitors in the observational model mapped into a new unit, amplifier, in the abstractional model.

As another example of specific mappings, we can generalize "location higher than" to "above" by:

Vx,y,t,t',l, bool

MAP({locationHigherThan(x,y,h,t,t',l,bool)|h>0 } (2.1a)

U {onTop(x,y,t,t\l,bool)})

= above(x,y,t,t',l,bool)

Note, both the observation and abstraction are sets, but we have used the abbreviations introduced above. As another abbreviation, in the future, we may write

MAP(p)=q

instead of

Vx,...a, ...t,l,bool 3x\...a'...t\ 1\ bool'

[MAP(p(x,..a_v..t,l,bool))=q(x',..a',...t',l',boor)]

if it is clear from the context *what the arguments of p and q are.*

A mapping is an identity mapping if

MAP(Situ)=Situ

A mapping is a null mapping if

MAP(Situ)=0.

Consistent **Mapping Set**

To define a mapping from one model to another model, we may use a series of equations like (2.1a). A sot of equations that collectively define a mapping is called a complete mapping set.[9]

The fact that a mapping forms a set of equations or rules poses a potential problem as they may be in conflict themselves. E.g., if we have a mapping set MAP such that

$$\mathrm{map}_1,\ \mathrm{map}_2 \in \mathrm{MAP} \wedge \mathrm{map}_1(p)=q \wedge \mathrm{map}_2(p)=r \wedge [q \neq r],$$

we get two different, possibly contradictory abstractions depending on which rule we choose to use. A situation like this is undesirable.

On the other hand, for most practically useful mappings, the same observation p may indeed map to different abstractions in a *different context*. Typically, one may have

$$\mathrm{MAP}(p)=q \tag{2.2a}$$

and for a function f,

$$\mathrm{MAP}(f(p))=Q \wedge Q \neq f(q) \tag{2.3}$$

at the same time. Or, put in another way,

$$\mathrm{MAP}(f(p)) \neq f(\mathrm{MAP}(p))$$

For example, in the case of our radio set, when we view a device as a whole the wires and components inside would not be reflected in a more abstract model. But the wires connecting devices will be.

Apparently, if a fixed order of applying mapping rules is enforced, we can always get the same abstraction from the same observation. But this has deviated from the requirement that the mapping to be functional (in the algebraic sense) rather than procedural (i.e., like parsing). But we point out that to handle the problem of (2.2a) and (2.3a), it is not necessary that we have a fixed order. In fact, if no other argument in function f is affected by the mapping, the order of application of rules (2.2a) and (2.3a) does not affect the final result.

We remark here that many mistakes made in writing descriptions are due to inconsistent or incompatible mapping rules. But we will not discuss this any further. In the future, we will always assume implicitly that the mapping rules given are a compatible complete mapping set, which means the order of rule applications does not matter.

Mapping Between Theories

The definition of mapping between theories follows strictly from the mapping of models. Namely, for two models OBS and ABS (observation and abstraction)

$$\text{if } \mathrm{MAP}(\mathrm{OBS})=\mathrm{ABS} \text{ then } \mathrm{MAP}(\mathrm{OBS}_{fset})=\mathrm{ABS}_{fset}.$$

[9]Note the word "complete" is used only in the sense that one is content with the rules one already has. It has nothing to do with the completeness of the resulting domain in whatever sense.

Similarly[10] we can have

if MAP(OBS)=ABS then $\mathrm{MAP}(\mathrm{OBS}_{closure})=\mathrm{ABS}_{closure}$.

We note that although a theory has logical consequences, the mapping as an operation does not act on the logical consequences, but only on the form of each formula. In general, a mapping for formula sets p and q does not meet the following:

if $p_{fconjunction} \equiv q_{fconjunction}$ then $\mathrm{MAP}(p_{fconjunction}) \equiv \mathrm{MAP}(q_{fconjunction})$.

Mapping between formulas or theories is not entirely new. As a matter of fact, deduction schemata are mappings between formulas. However, one may still question why we would use the notion of mapping rather than the notion of producing a new theory by expanding an old one, as it is usually done in mathematics, e.g., by adding Peano axioms to standard logic we got a theory for natural numbers.

There are two reasons for not doing that. In the first place, there are technical difficulties. For example, we do not know how to make the resulting new theory not have some of the old axioms, i.e., make the new theory "smaller." (We know most abstractional theories are smaller than their observational counterparts.) Another technical reason is that, comparatively speaking, functional mapping is a very restricted form of transformations and we may want its fine mathematical properties. We do not want *arbitrary* axioms to be used for defining new theories.

But a more important reason is that there is a feeling that to view it as a mapping is more consistent with the way the new theory is generated. Namely, it is based on another *mapping*, mapping of models. Therefore, this view is more intuitively appealing.

Practically, it is often simpler to put theory mappings into some axioms as though there is a larger theory that contains both. This is because most we do with formulas are deductions, and axioms suits deduction schemes better than equations. Also, theoretically, there might be a good way of reconciling the difference or establishing their equivalence. We note that the actual way of dealing with them is just a minor point. What is important is that the relation between the two models is a mapping and the relations between the two theories are *based* on this mapping.

2.4.3 The Definitions of Properties of Mapping

The soundness, completeness, and other properties of a mapping are defined in what follows.

To help illustrate the points, figures 2.4 to 2.7 provides a schematic guide. Here we have two levels and many "clouds." The two levels are observations and abstractions. Each cloud

[10]If you want to be really careful, you may use MAP_{theory} instead of "MAP." But there is no need for it for our discussion.

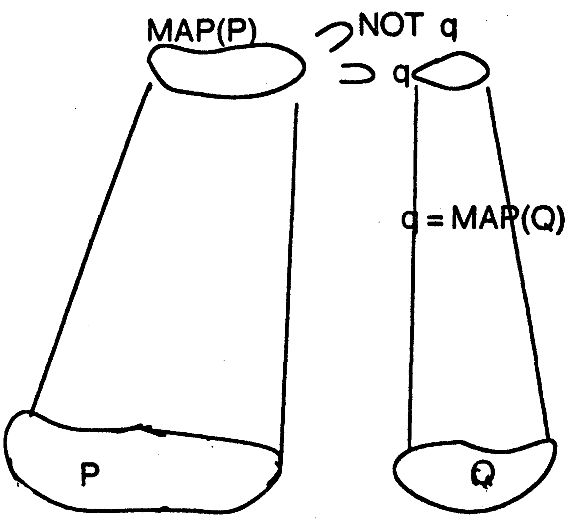


Figure 2.4: The Soundness of Mapping

is the theory that corresponds to a domain. Question marks signal a particular problem in the picture.

Since the reader will need to constantly refer to these pictures in order to follow discussions, it is a good idea to familiarize oneself with them from the start.

Assume that OBS is the domain containing the complete observation from which we select a part to describe or to model. $OBS_{closure}$ (its equivalent formal theory), as explained earlier, is assumed to be sound. Moreover, it is assumed to contain all the sentences derivable from OBS_{fset}. Practically, there is no sense to formally define any kind of completeness for $OBS_{closure}$.

Assuming Obs, OBS are situations or domains and

$Obs \subseteq OBS$, $Abs=MAP(Obs)$,

consequently

$Obs_{fset} \subseteq OBS_{fset}$, $Abs_{fset}=MAP(Obs_{fset})$

we give following definitions:

Definition of Soundness. *mapping MAP is sound, if and only if $Abs_{closure}$ is sound.*

Referring to fig. 2.4. we expect $Abs_{closure}$ does not entail both P and $\neg$P, if $Abs_{closure}$ is sound. E.g., in an abstractional theory of our radio set, no matter how the mapping is constructed, this property requires that we do not get a contradiction. For example, we do not infer both that the speaker is generating sound output and that it is not, from the working of the amplifier and the not working of the antenna.[11]

[11]Depending how the term "sound" should be interpreted, a sound can be considered as being generated

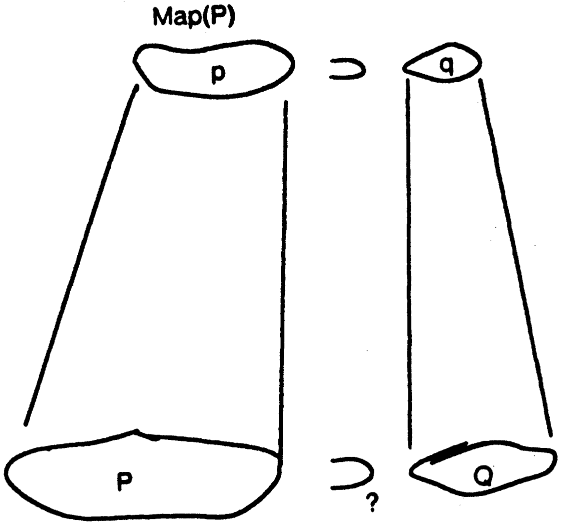


Figure 2.5: The Relevance of Mapping

The notion of soundness is rather weak. We define the relevance of a mapping as follows:

Definition of Relevance. *mapping MAP is relevant if and only if*

$$\forall \text{abs}_{fset}\ [\ \text{Abs}_{closure} \rightarrow \text{abs}_{closure}] \rightarrow$$
$$[\exists \text{obs}_{fset}\ [\text{MAP}(\text{obs}_{fset}) = \text{abs}_{fset} \wedge \text{Obs}_{closure} \rightarrow \text{obs}_{closure}]]\ ;$$

The relevance axiom hints that the logical consequences of an abstraction (or the mapping of a piece of observation) should be confirmed by other observations. And this is basically expressed in Fig.2.5., where p finds P, its inverse mapping, in Obs_{fset} and P is the consequences of $\text{Obs}_{closure}$.

Again, in the case of our radio, this could mean that if we can infer, at the level of abstractional model, the working of the amplifier from the facts such as that the speaker is generating sound output, then we should be able to identify a particualr state of signal input and the network of connections of transistors, wires and so on in the observational model, such that their mapping in the abstractional model would be a situation of the amplifier being working. We note that, in general, there could be many combinations of connections and signal states that map to a situation of the amplifier being working. There could be none, too. The relevance property requires we can find at least one such situation so that the inference made at higher level makes sense at the lower level as well.

We note the existential quantifier in the relevance definition can not be replaced by a

when the amplifier is working or only when the radio receives broadcast. In the former case, there is only background noises, they are meaningless but they are still acoustic waves.

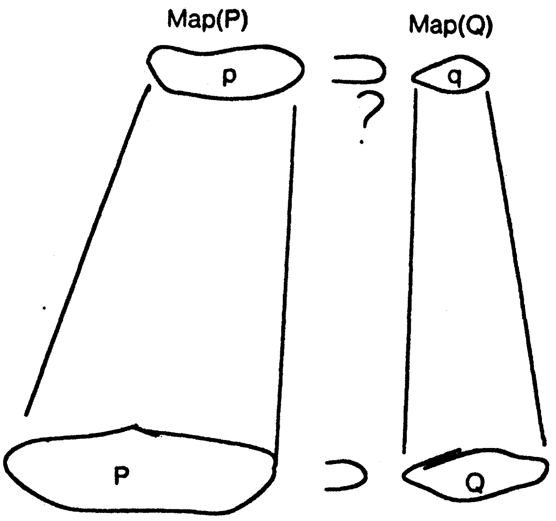


Figure 2.6: The Consequence Preserving of Mapping

universal one[12] because very often the mapping is many to one. For example, a relation "above(A,B)" may be the result of "A is 4 feet higher than B" or "A is on top of B" and so on. But, of course, only one state of A and B is possible.

Probably the most basic thing we can expect from a mapping is to be able to infer something at a higher level if its counterpart can be derived at lower level. Referring to figure 2.6, this requires that if Q is implied by $Obs_{closure}$ then $MAP(Q)_{closure}$ should somehow be derivable from $Abs_{closure}$. This is defined as follows:

Definition: Consequence Preserving mapping. *mapping MAP is consequence preserving if and only if*

$$\forall P,Q \subset OBS_{fset}\ [[P_{closure} \rightarrow Q_{closure}] \rightarrow [MAP(P)_{closure} \rightarrow MAP(Q)_{closure}\ ]]$$

Here all the mappings can be empty. And we note that this is natural. This corresponds to the fact that we may decide to take one or more factors of a fact for granted and investigate the rest. In the case of our radio set, this would be the converse of the property of relevance. That is, if we know a particular connection and a particular signal pattern give rise to a particular pattern of some performance, then, in the abstractional model, their mappings should preserve this implication relation.

Definition of Completeness. *A mapping MAP is complete, if and only if*

$Abs \neq \emptyset \wedge$
$[Cardinality(Abs)=1 \vee$

[12]With "$\wedge$ " changed into "$\rightarrow$" correspondingly, of course.

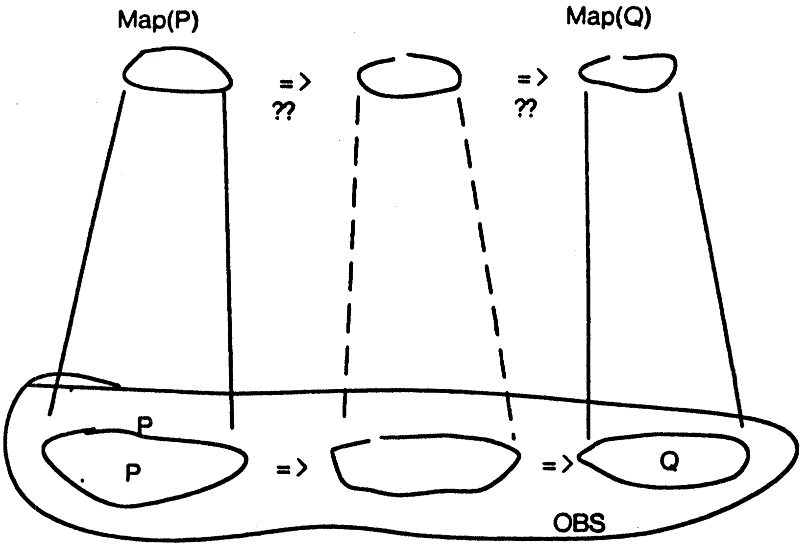


Figure 2.7: The Operational Completeness of Mapping

let P,Q $\subseteq$Obs,
if [affect(P_{fset},Q_{fset}) $\wedge$ $\exists$p,q[p=MAP(P) $\wedge$ q=MAP(Q)] $\wedge$ p,q$\neq$ $\emptyset$]
then $\forall$R [if (affect(P_{fset},R_{fset}) $\wedge$ affect(R_{fset},Q_{fset}))
then MAP(R) $\subset$Abs $\wedge$ MAP(R)$\neq$ $\emptyset$].

The predicate "affect" is defined in the section on event abstraction of Chapter 6. For sets of formulas the predicate "affect" is generalized to take the conjunctions of the two sets as its arguments. Intuitively, "affect" means that the interpretations of the formulas, i.e., the events or relations, influence each other and probably causally relate to each other.

Referring to figure 2.7, this says intuitively that if there is a set of relations or events in the observational model that *influences or causally relates to* another set and both have a mapping in the abstractional model, then anything on that causal chain should also have a non-empty mapping.

In our radio set example, this property means that, as we mentioned earlier, if an object or event is directly or indirectly related to the sound output of the radio, then it should be included in the abstractional model, e.g., the wire that connects the speaker and the audio amplifier. On the other hand, the color of knob need not be. We note this abstraction need not always be explicit. As a matter of fact, we may decide to abstract the wires and the speaker together into a "sound unit". In that case, the sound unit would have to be considered as directly connected to the audio frequency amplifier. In this way, nothing on the causal chain is lost.

The implications of this axiom need some explanation. It is important to note that all of the elements of OBS are individual relations or events. They can not be individual objects. Two individual objects can affect (in the normal sense) each other in many ways but when two events are said to affect each other, the way of affecting is completely determined by the type of events that have happened.

2.4.4 Discussion of The Definitions

All these definitions have some motivations between the lines and need explanations.

At first glimpse, some of the properties seem to be derivable directly from others. One such conjecture is that, $0BS_c/_O3ure$ being sound, if the mapping MAP is relevant then the resulting abstraction would be sound.

We can show the falsity of such a conjecture with a simple counterexample.

E.g., we can have a theory mapping[13] that ignores time, e.g.,

Vt MAP(P(t)) = p[14].

Now it is possible to have a domain OBS, and consequently a theory *OBSdoaure*, where for any predicate class P[15] there is t and t' such that predicate formulas P(t) and -~vP(t') are true in the observational theory. Then we will have both p and -ip in the abstractional theory. And this mapping is relevant because for any thing that can be derived from $Abs_c/_{O5tire}$, say, p and -ip, we have

VP 3t,t' {P(t)>, {-.P(t')}6$0BS_{/\#}$* A

MAP(P(t))=p A MAP(-iP(t'))=-«p.

This may seem simplistic. However, the reason a piece of Fortran program such as

```
x = 7
...
x = 5
```

seems absurd to someone unfamiliar with the language is exactly because he uses (or the Fortran notation suggests) a mapping that neglects time.

Ignoring spatial differences can result in similar unsound mappings. One way this can occur is if partial descriptions are allowed to be used to identify objects. For example, we may talk about a person named John, sitting in a MacDonalds and eating hamburgers. Now if it happens that Mr. John White is sitting at place A and Mr. John Black is sitting at place B and both are eating hamburgers we will get another inconsistent theory for a consistent observation.

---

[13] Any theory mapping conies from a corresponding model mapping but this example is so simple that we safely assume there is a model mapping existing.

[14] Presumably predicate P and p may have other arguments. Assume now they are implicit.

[15] The notion of predicate classes will be explained in the chapter on time.

Furthermore, one is tempted to think that since both facts and implications in $\text{Abs}_{closure}$ are generated by a mapping, everything they infer should be the mapping result of something implied at lower level. In other words, a mapping is automatically relevant.

However, because the resulting implications may increase their "implication power" through generalization. This is not the case. As an example, we again refer to a mapping that ignores time, say,

$$\text{MAP}(p(t,t',l,bool)) = p,$$

and for the same reason ignores any restriction on time, i.e., for any expression f

$$\text{MAP}(f(t,t',t''...)) = \emptyset$$

if t, t', t", as temporal variables, are the only arguments to f. A time dependent relation, say,

$$p(x,y,...a,b,t,t',l,bool) \tag{2.5}$$

will now be

$$p(x,y,...a,b,l,bool). \tag{2.6}$$

If (2.5) is in Obs then (2.6) will be in Abs.

Now in terms of their corresponding predicate formulas if, say,

$$\forall x,y,t,t' \; [p(x,y,...a,b,t,t') \wedge \text{evenNumber}(t)] \rightarrow q(x,y,...a,b,t,t')$$

is in $\text{Obs}_{closure}$, then

$$\forall x,y \; p(x,y,...a,b\,) \rightarrow q(x,y,...a,b\,)$$

would be in $\text{Abs}_{closure}$. From (2.5), q(x,y,...a,b) will then be in $\text{Abs}_{closure}$, which we would happily accept. On the other hand, if

$$p\,(u,v,...c,d,t_1,t_1',l_1,bool_1) \tag{2.7}$$

is in Obs, then predicate formula

$$q(u,v,...c,d)$$

would also be in $\text{Abs}_{closure}$, even if t_1 is not an even number.

In reality, the relations among these properties are rather complicated. And any general treatment is difficult. There are two origins of the difficulty. First, the mapping can be arbitrarily complex and "bizarre." Second, the mapping is a many-to-one mapping. If a proof relies on going back from abstraction to observation, it will have the multiple values inherent in the inverse mapping to handle.

2.4.5 Proving Properties of Example Mappings

If *as the result of a model mapping* the theory mapping has the following characteristics, then it is a *logical connective preserving* mapping:

1. If [P A Q] is in *Obs$_c$i$_{osure}$* then MAP(P A Q)≡[MAP(P) A MAP(Q)] is in Abs$_{d<wurc}$.

2. If [P V Q] is in 0bs$_c$/$_{O5tirc}$ then MAP(P V Q)=[MAP(P) V MAP(Q)] is in *Abs$_c$i$_{O9ure}$*.

3. If -iP is in *0bs$_c$i$_{O3ur}$e* then MAP(^P)≡[^MAP(P)] is in Abs$_c$/$_{O3ttre}$.

This mapping is not the identity mapping because one can still do tricks to atomic formulas, for example, replacing

Greater(height (x) ,height (y))

with

TallerThan(x,y),

which is a step of abstraction, although not a very useful one. To make a practically useful abstraction, many mappings of the form of

MAP({p(x,y),q(x,y)})=r(u)

are necessary.

An *object identity preserving* mapping has the characteristics of

MAP({P(x,y,..t,t\l,bool) | t,t\l, bool})= {p(x,y,...ti,t$_l$M\boai')| tx^'^booi'}

where P and p can be *arbitrarily complex* situations. The parameters, the times, and the locations can all change. The requirement is that objects remain the same. We remark that this requirement may be considered inappropriate from two ends. If an abstraction tries to aggregate, the components are often replaced by their composed whole. On the other hand, the "absolute" identity may not be necessary for some occasions. For example, in a store where people shop with cash, as a first order approximation, the store owner can view any *person* coming in the door as a new *customer* and treat him/her accordingly.

Order preserving mapping is defined by following mapping rules and formulas:

MAP(p(x,y,z...,t,tM,bool))=p(x,y,z...r,r\l,bool)
if MAP(p(x,y,z...,t,t',l,bool))=p(x,y,z...r,r',l,bool) A (2.10)
MAP(q(x,y,z...,t$_1$,t$_1$\r,boor))=q(x,y,z...,r$_1$,r$_1$\r,boor) A t<t$_x$
then *T<TI*

where

T<T' A n<ri' (2.8)

and

r,ri,r',7y GT

and T satisfies the following formula for time points t,tjthat appears in a relation or event in Obs:

3s [t<s<ti]->3<TeTr≤a≤ri (2.9)

If, in an observation Obs, temporal variables only appear as beginning and ending time points, the mapping will be a one-to-one mapping. For the corresponding theory mapping, since all the facts are mapped into the abstraction, the implications are also mapped into the abstraction (as deductive closure), it should be obvious that this mapping is sound and complete. As a matter of fact, it is sound and complete even without the requirement that r and *T*± be ordered because the temporal variables would be viewed as ordinary variables.

However, in our case, since the new temporal variables are intended to indicate the beginning and end of the truth of a proposition, the requirements (2.8) and (2.9) are important. The global order (2.10) is necessary if we would like to see the new model represent the behavior in the same order as in **Obs.**

We note the above described mapping is just what we usually do in a framework of discrete time. And we have noticed the restrictions due to the way our mapping is defined, namely, that any time variables not in those positions may render the mapping unsound. For example, if a relation or event or rule depends on the actual time span of an event, this mapping will not be able to handle it. On the other hand, if only relative ordering **is** concerned, this would be the right level of abstraction. As a matter of fact, a sequential Pascal program can be viewed, among other things, as an order preserving mapping of all actual runs of the program.

An Analysis of the Examples of Pathological Mappings

Returning to our motivating examples in an earlier section, the event composition problem can be put in mapping terms as follows. Assuming we have three consecutive events which have certain consequences, a normal mapping would be as follows.

Let obs={p(x,y_r.ti,t_2,l,booli),r(x,y,...,t,tt,l, bool), q(x,y,...t_1',t_2',l,$bool_2$)}

where

ti<t_2<t<ti'<t_2'

MAP(obs) = E(x,y,tx,t_2',l,T).

We now assume there is the following implication in $OBS_{c|odwrc}$:

Vt,t' 3e,s,u,t+,t++ (2.16)

[t<t'<t+<t++ A A r(t) ->s(u,t) A -i[q(x,y,...t') ->-.s(u,t')] A

s(u,t+)->e(u,t-f) A e(u,t+)->-ir(t)].

The intended meaning of this condition is that the relation or event r will afiFect a relation s; the happening or holding of event or relation q will not affect s; s will affect the occurence of event e; and the occurence of event e will make r no longer true and, therefore, the chain will not recur. As a result of (2.16), we will have an observation $obs(t_1,t_1')$ for some t_1,t_1' and an event $e(u)$ such that

$$obs(t_1,t_1')_{closure} \rightarrow e(u)$$

Since obs would be abstracted into a whole E, to reflect the above implication this mapping would introduce in the abstractional theory:

$$E(t_1,t_1') \rightarrow e(u). \tag{2.16}$$

These normal mappings would have all the fine properties. The proofs will not be given here.

However, if we allow the "sharing of component," there would be a mapping [16], which is like the following.

$$\text{Let } obs_1=\{p(x,y,..t_1,t_1',l,bool_1),\ q(x,y,...t_2,t_2',l,bool_2), r(x,y,...t,tt,bool),$$
$$p(x,y,..t_3,t_3',l,bool_3), q(x,y,...t_4,t_4',l,bool_4)\}$$
$$obs_2=\{p(x,y,..t_1,t_1',l,bool_1),\ q(x,y,...t_2,t_2',l,bool_2), r(x,y,...t,tt,bool),$$
$$p(x,y,..t_3,t_3',l,bool_3), q(x,y,...t_4,t_4',l,bool_4), r(x,y,...t',tt',bool')\}$$

where

$$t_1<t<t_2 \wedge\ t_3<t<t_4 \wedge$$
$$t_1<t'<t_2 \wedge\ t_3<t'<t_4,$$
$$MAP(obs) = MAP(obs_1) =$$
$$\{E_1(x,y,t_1,t_2',l,\ T),\ E_2(x,y,t_3,t_4',l,T)\}.$$

As for the implication rule in the abstractional theory, there seems to be only one possibility, that is

$$\{E_1, E_2\}_{closure} \rightarrow e(u)$$

or

$$E_1 \wedge E_2 \rightarrow e(u).$$

In general, we can not have,

$$[E_1 \wedge E_2] \rightarrow [e(u) \wedge e'(u)]. \tag{2.18}$$

This means that for a situation containing obs_1 the mapping is not consequence preserving.

Of course, one can manage to make (2.18) true in the abstraction. (Remember, the mapping can be arbitrary.) For example, I can simply have obs_1 map to both $\{E_1, E_2\}$ and $\{e(u), e'(u)\}$. In that case, we will infer something in the abstraction which is not derivable at the observation level. That is, the mapping would not be relevant[17].

[16] This mapping can not coexist with the above one. In that case, the two mapping equations will be inconsistent. We will have a problem, but of different sort.

[17] We note that the mapping is otherwise logic connective preserving and identity preserving. This is due to the fact that formulas in both theories are associated with unique objects (x and y in the above mapping) and beginning and ending time points.

In the case of the example motivating the definition of completeness, let us just point out that tea serving does not enable any event, but the event ordering will result in an order that causes the chef to cook and the waiter to serve.

Before leaving this chapter, we make a note about the notation. It has been painful to burden our readers with expressions with so many arguments and subscripts, but it was necessary to clarify the concepts. Now that the task has been accomplished, we will use a much freer notation. In the future, we will not subscript symbols for distinguishing entities in models or theories. The reader should be able to recognize them from the context. The arguments of formulas will often be omitted where there is no danger of confusion.

Chapter 3

Categories in Operational Domain Model

There are three major categories in our domain: object, relation, and event [Barwise and Perry] [Wolterstoff].

3.1 On Objects

3.1.1 Basic Concept of Object

"What is an object?" For a question like this, we can expect thousands of different answers from people. Philosophers, logicians, programmers in object-oriented languages, and "ordinary guys" on the street all have their typical answers.

For us, *objects* are "things" or, to be exact, models of "things" in the real world, individual or collective, and their abstractions. But this is a bit vague. To clarify it we will do two things: First, compare our view with others' and second, try to formalize.

We start the comparison with the philosopher's notion of object.

Unfortunately, our philosophers do not quite agree on this issue. For realists, the chair I am sitting on is an object. But for Bishop Berkeley, it is nothing but "a compound of sensations." As a matter of fact, this controversy is at the very heart of metaphysics. A philosopher once declared ontology, supposedly the science of pure being, as a "mistake," since "there is no such thing as pure being."

Even if we agreed on the existence of the chair I am sitting on, be it reality or illusion, we still get another question. As [Wolterstoff] asked: "A bird is a thing," but "the bird flies" "is flight a thing?" An orchestra is a thing. Is the symphony they play a thing? An honest person is a thing. Is honesty a thing?... Philosophers are divided here, too.

Our theory does not address these issues directly. We will take for granted that there are such things as objects. And we assume that some of them can be directly observable. If it can be observed, then it is in our observational model. As for the nature of these "objects," it is beyond our current concern. In other words, we find our objects in the observational model and we do not care how they get **here.**

Objects: Models of "Real Things*

The way an observational model relates to "true" reality needs some explanation. An easy mistake to make is to take the model as some kind of *representation.* The model being formal makes one even more suspicious about this. But this is not necessarily the case.

A model is an abstract entity. However, as we know, the degrees of abstractions can vary. Relative to the infinitesimal in calculus, a dog is something very concrete. However, a dog, say, Fido, is itself an abstraction, if we "realize" that it is "but" a huge bunch of particles and intervening space. Our model is "abstract," just as Fido is "abstract." The object Fido is not the symbol "Fido," nor the picture of Fido. It is everything we have in our observational model about Fido.

Linking the "object Fido" to our observational model does not equate it to observations per se. In general, observations are the results of psychological processes. But objects in our model are abstract entities, independent of a particular observer. The word "observational" just reminds us of its origin and suggests the grain size of this level of abstraction. It does not entail any idiosyncrasy.

Furthermore, we warn that the model Fido is not the real Fido. The model Fido does "bark,"[1] as we have seen Fido bark. However, there are so many things we do not know about Fido; it is there that our model Fido becomes incompetent.

A reader knowledgeable in situation semantics [Barwise and Perry] may find that this notion of ours is very close to the notion of *individuals* in the theory of situations. The reader's impression is justified. In a way, our observational model is similar to what they call abstract situations, both being formal structures. Moreover, the primitives from which the formal structures are built are "real things," as they put it. However, these real things are not "as real as" Reality. Barwise and Perry "view real situations as metaphysically and cpistcmologically prior to relations, individuals, and locations." We believe there is something that is behind these "real things" but we do not commit ourselves metaphysically any further.

[1]**In the world where the model Fido is, there is a model for event "bark."**

So, we seem to be closer to the logician's view of objects, who assign terms of their logical theory to objects in their models. Yet we differ from them on the other end. To qualify for logician's model, an object can be any thing in any possible world. We, on the other hand, require the model to be meaningful to the world we live in and an object must be an abstraction from that world.

Objects: Not Entities in Computers

"Object" is now one of the most frequently used words in object oriented programming. There, an "object" is a mathematical or computational entity. It refers to things that exist inside computers or their abstractions. Metaphorical names are borrowed from "real" objects, such as "student," "employee" etc. But they are just synonyms of records or some other data structures. This is easily revealed by how a "student" "graduates"- He passes a message which is statically or dynamically looked up in the message list of relevant classes.

We, of course, intend to make use of our object notion in computation. But the distinction between the representation of an object and the object itself should always be made clear. When a student graduates, the situation changes. The secondary concern is which changes to represent and how.

It is interesting to compare our "object" and what people refer to by nouns. In a natural language, proper nouns are used to refer to individuals. Common nouns for classes. Definite references for individuals or classes. Not only there are nouns for physical objects but also, through nominalization, there are nouns for abstractions like poem, sonata or flight. It seems natural languages are well prepared for the model we have in mind.

3.1.2 Object Categories: An Informal Account

We propose to make distinctions among the objects in a model, relevant to the domain of interest.

First there is a contrast between the class of physical objects and abstract objects. A physical object has a *unique spatial location,* is *directly observable,* and has an *identity* associated with the location. Here, by "location" we mean a set of points in the three dimensional space occupied by the object. We usually talk about discrete space, in that case, a point in space is a very small cube. We cannot always identify an object with a location, i.e., a set of points, since we have cases where the sets of points of two objects intersect, but for every physical object there is a location associated with it. Contrary to physical objects, an abstract object exists by virtue of being the abstraction of certain set of other objects and obtain its identity from the set of objects and the abstraction function that define it. For example, the meaning content of copies of a book with the title "Hamlet" is an abstract object, a piece of Shakespeare's tragedy. But it can be as well viewed as a piece of English text depending at which abstraction level one would like to see it.

As noted from start, we are interested in describing systems involving human activities. This implies that cooperation and communication will hold significant position in this class of systems. Based on this particular interest, we further divide the class of physical objects into subcategories including concrete objects which do not designate any symbols, physical forms of information objects which do, and agents. The key concepts here are information objects and agents.

Information Objects

Information objects are objects that carry information. The particular importance of communications in systems and the conceptual distinctiveness of these objects make them a special category.

An information object is often the combination of form and content along with the type of communication functions it serves to accomplish. For example, a document may be in the form of a slip of paper with some writing on it. Depending on the document being a bill or receipt it can either mean that one has to pay certain amount of money or that one no longer need to.

To be conceptually clear, we distinguish three kinds of objects: physical form, abstract form and abstract content of information. A physical form, being a physical object, designates a symbol which is the abstract form. For example, a copy of a book as opposed to the text of the book, the magnetic state of a block of computer memory as opposed to the sequence of bytes it "represents." In turn, a symbol will always denote or be interpreted as something which we might call a concept, i.e., its intension or abstract content. For example, in a mail order business, a customer produces certain string of letters which may be the physical form of a symbol, say "lamp." The concept of the symbol, say "lamphood" or the concept of the lamp class, would be conceived by the clerk and the warehouse workers. There, it is finally realized as a real physical object, a lamp.

It is important to note that there is a fundamental difference between something, say a lamp, and another thing which symbolizes the thing, say an order for the lamp. Natural languages often blur this difference. For example, it is perfectly all right for one to say that "a customer makes an order from a catalog and a mail order company ships a package containing the order to the customer." However, to specify the business activity precisely, one has to make sure that what the customer sends in is some symbol while what he receives is what the symbol stands for: one is a piece of paper, the other is the lamp.

In a system having many agents, the agents communicate with symbols. The symbols are normally later realized as other objects or events. This is the function of symbols. An information object is said to be realizable if its potential realization exists in the same system. For example, the lamp order in a store is a realizable information object. So is a command in an operating system. A file in a file system is usually not realizable while a directory entry is.

Concept: Abstract Content of Information Objects

Our notion of concept is similar to that in [Brachman]. We stress, however, that operationally a concept is a highly abstract entity that has its realization as other objects (physical and abstract), relations among objects or properties of objects. For example, a measurement, a class abstraction, colors, numbers, events, and so on, can all count as abstract objects.

In many systems, a concept will eventually be realized as objects, relations and events etc. In the particular case in which an object already exists when its symbol is mentioned, we can also say the symbol expresses the object, and the concept denoted by the symbol is interpreted as the object.

Agents

A thorough discussion of the concept "agent" will be given in a later section. Here we just point out that agents are the only class of objects that can communicate and initiate actions on their surroundings. To be able to communicate, an agent should be able to recognize and manipulate symbols. An agent initiates a planned action for achieving a certain goal.

Figure 3.1 shows a map of various categories of objects in our model.

Object Classification

Some members of a collection of objects share common attributes. These common attributes define an abstract entity which is called a class. This class concept has been proposed by many authors within computer science, starting with the design of the language Simula. (See [Birtwistle, Dahl, Myhrhaug, and Nygaard], [Goldberg and Robson]).

A class is the intensional content of the collection of objects. A member in the collection is said to be an instance of the class or to be an instantiation of the intensions that the

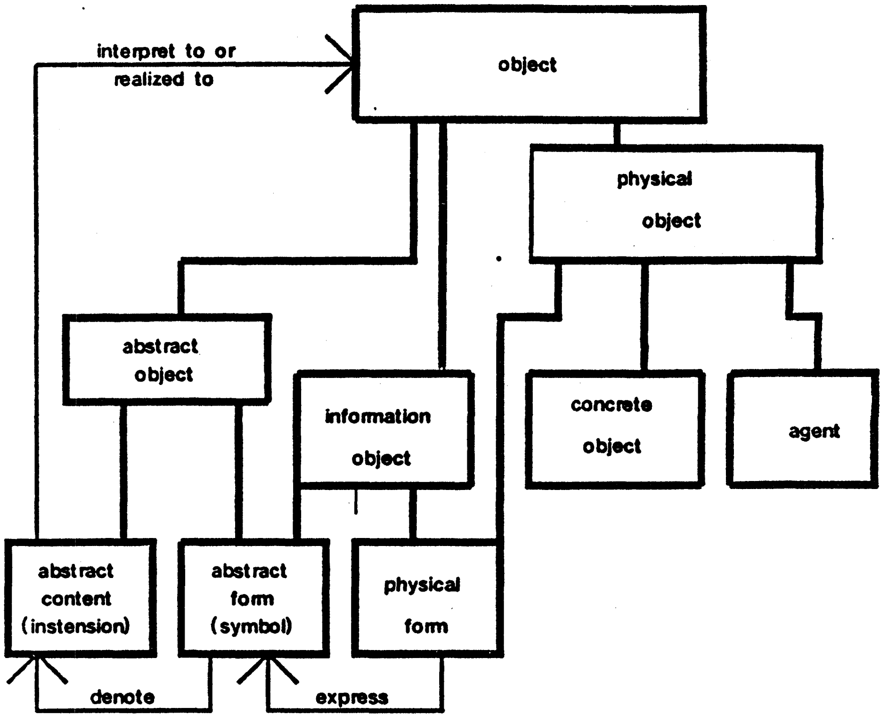


Figure 3.1: Object Categories in the Operational Domain Model

class stands for. For example, a physical copy of a book is an object, while all the things that are books define the class, book. Similarly person is a class of objects while John or Jack would be an instance of the class person.

Just as there can be a subset relation between two sets viewing the extension, there can be a subclass relation between two classes. For example, the class student is the subclass of the class person.

We note that not all sets have corresponding classes. A set, as a mathematical structure, may include things having little in common and we may be reluctant to think there is a class that corresponds to it. However, a class can always find a corresponding set (possibly empty). In the future, we will take the correspondence of the extensional notion "set" and intensional notion "class" for granted whenever we have defined a class.

3.2 On Relations

The Intuitions about Relations

Another basic category of observations is relations. Like the object, a relation as it stands is abstract in the sense that it has been singled out from the great flux of Reality but also concrete in the sense that it is not a formal representation per se.

A clear distinction can be observed between the notion of relations that exist in logic and those that appeal to people's "common sense." In logic, a relation is just a set of tuples. And relations can be constructed from relations by manipulating these sets at will. For example, if P(x) and Q(y) are two relations then

$$R(x,y) \equiv P(x) \vee Q(y)$$

is a legitimate new relation. The commonsense notion of relation is much more restricted though. For instance, in the above case, people do not accept R(x,y) as a relation relating x and y. Although, of course, we do agree on it being a predicate with a well defined denotation.

This feeling is justified. As a matter of fact, we find relevant remarks in situation semantics, where relations are declared to be "not words or sets, but properties of the kind recognized by human beings." [Barwise and Perry] A whole theory will be devoted to this issue later.

Relations vs. Events

Relations are different from events. Most relations are something like "states," i.e., they can be observed and measured instantly, as opposed to events which are accompanied by state changes and happen over a period of time. To observe an event or to make a judgment that an event has happened takes as long a time as the duration of that event. For example, a block A being on top of block B is a relation and the move that put A on B is an event.[2]

A relation may hold for a period of time. However, throughout this time period, "nothing" happens as far as this relation is concerned. An event, on the other hand, is characterized by changes of relations. Usually, for the duration of an event, the relations it changes will be taken as undefined. We will have more to say about this when we come to the section on events.

In the future, we will use an n-ary predicate to represent a relation involving n objects and call the relation an n-ary relation. A relation can be functional. In that case, a function (in the logic sense) may be used for its representation. A relation represented by a unary function is also called a property.

[2]Those relations that cannot be instantly observed are really different versions of judgments on events. They are historical relations and will be briefly explained later.

3-2.1 The Taxonomy of Relations

The world of relations we observe can be classified along many dimensions. Some are easily formalizable. For example, they can be grouped according to the number of objects involved, as we did a moment ago. Similarly they can be grouped by temporality. Some will be true all the time, e.g.,

equal(sum0f(l,l),2)

or more familiarly,

1+1=2.

Some will not be that fortunate. For example, for any place P on Earth (except near the poles),

Sunny(P)

will not last longer than or equal to 24 hours.

We would like, however, to start our classification from a conceptual dimension, a dimension where the nature of the relation itself is the line of demarcation.

Causal vs. Non-causal Relations

We differentiate relations like "taller than," "to the left of," etc. from relations like "connected to," "supported by" (in the blocks world), etc. We claim a relation in the first category does not necessarily entail any causal effect. As a matter of fact, any two objects having height can be related by the relation "taller than." On the other hand, the latter relations represent the way objects act upon each other. These are *causal relations.*

It is true that when some changes happen to the objects, both kinds of relations change. But we can see there is a degree of directness. When a child grows in height, there are an infinite number of things that once were taller than him that now fail to be so. All these falsified relations, however, derive from that common cause. Namely, the body of the child has changed. This gives us grounds to believe that relations can be either "derived" or "direct." "Direct," in the sense that it can not be inferred from external relations with other objects or among other objects. It can only find its grounds in the relations between its own parts, if it can be decomposed, or it is simply an inexplicable fact, primitive for our observation.

There are at least two basic classes of causal relations: physical and social. "Connected to" and "supported by" are examples of the former. "Own," "owe," "superior in rank" are examples of the latter.[3]

[3]It is likely that mental states have some causal power too. Nonetheless, we do not deal with them in

Interactions

We call physical causal relations *interactions*. The word "interaction" is a heavily loaded word. The reader should take note that, in the future, we may use the word in its nontechnical sense, when no confusion results.

Interactions can be easily characterized and identified. They are always based on direct physical contact. (The contact can be an electromagnetic field or any interactions modern science finds evidence for.) In this way, it is also localized.

Social Relations

Social relations are much more difficult to put a finger on. For example, if someone possesses a certain amount of money, as a sales clerk in a grocery store does, the relation is *more than the spatial relation that the money is accessible to him in some way.* If you try to take the money from him by force, you break the law or you are doing something bad, if there is no law written yet. Similarly, if we say someone is occupying a seat in a movie theater we expect other people will not take that seat, *even if the person is not sitting there at the moment.* And it is obvious that there is no law to break if someone else takes the seat.

A social relation is definitely different from a relation in general, even if they have the same consequences in certain settings. For example, if two cars arrive at an intersection at the same time and one car is to the right of the other, the latter is supposed to wait for the former. Now, for the drivers, recognizing this spatiotemporal relation is sufficient for them to make the right decision. But that relation alone, as we all know, is not the cause of their behavior. The fact is that that relation entails the "right of way" for a driver and it is that right which allows him to go ahead. Similarly, someone may occupy a seat by putting a handbag on it. Is the spatial relation between the handbag (or any other object, for that matter) and the seat the source of the force blocking anyone else? Of course, not. The spatial relation only implies some social relation that exerts the real force.

Intuitively, social relations are legal or moral bindings among objects in a society. They are often expressed by explicit speech acts. Laws are typical examples of this. But in general, a social relation is not necessarily the result of a speech act, at least not in its ordinary sense. For example, there is no law that specifies how much care parents should give to a child. (Although there might be a law specifying how little.) Yet there are many things parents have to do according to the custom or culture they accept. We simply point out that social relations establish themselves in the activities going on in societies. Speech

our model.

acts are just part of the communication in a society and communication is just part of the activities in **general.**

Abstract Relations

We call the rest of relations non-causal relations *or abstract* relations. Abstract relations can be spatial relations, temporal relations, mathematical relations, historical relations, and abstraction relations. Separate sections will be devoted to temporal relations because of their extreme importance. We discuss the others in what follows.

Spatial relations can be defined on the basis of locations. As a matter of fact, when we talk about spatial relations between objects we mean the relations between the locations of the objects. Quantitative spatial relations involving our everyday activities can be handled successfully by known mathematics and we will not spend our time on them. Qualitative spatial relations are one of the most important human cognitive dimensions. Some typical relations such as "at," "on," and "with" are basic primitives in our representational framework. But their meanings are formally defined rather than directly taken from their use in English which is often very fuzzy.

Most of what we read about in mathematics books are *mathematical relations.* Simple examples of these are "=," ">," find "<," but the relations can get very complex, such as Bessel functions or Taylor series. The mathematical relations among real world objects (not their numerical abstractions) are relatively simple and not the salient problem. Therefore, we will not discuss this type of relation any further.

However, the combinations of spatial, temporal and mathematical relations may form an interesting class of relations: measurements. An measurement relation amounts to the comparison of the attributes of a "standard" object with that of the object to be measured. For example, "the length of the cord is 3 feet" or "the baby weighs 26 pounds" are measurement relations. They are important, as they are the basis for our quantitative knowledge of the world.

Historical relations are defined by what happened or what was true in the past. They serve to relate the present to the past or to what is simply a record of the past. The defining-past relations or events could be of any categories themselves. For example, one's age is a relation between the time one was born and the present time. Being an ex-employee of a company is a relation that records a past relation. Historical relations may or may not have any social consequences depending on the culture and the actual situation.

Abstraction relations relate something and its abstraction and vice versa. For example, the name of something and the thing, part of a device and the device, or event class vs.

its generalization, e.g., hitting versus touching. Because abstractions often happen across levels, it is not sensible to assume that there can be any causal links between things standing in abstraction relations. For example, in ordinary language we may refer to part of a device as causing the whole device to malfunction, but what we mean is that the interactions between that part and *the other parts* are the cause. A part cannot interact with the whole, simply because it can not interact with itself.

Things standing in generalization or classification relations(two types of abstraction relations), are the subject of property inheritance, which has become a key feature of object oriented languages and has been talked about extensively. We will not go into this any further except to point out that all types of abstractions entail some constraints that can be exploited by computational systems.

3.3 On Events

Event as a Basic Category

If relations are static accounts of the world, then events are meant to account for the dynamic side of the world. By saying that some event happened to an object, we mean there is some change in the object during or right after the event. An event can be instantaneous, as in the case of flipping a switch or hitting a keystroke, or it can be prolonged, as in the case of debugging a program or writing a thesis.

Be it instantaneous or prolonged, an event happens over time. We adopt a discrete system of time.[4] As remarked in an example of mapping in the last chapter, since the number of events concerning us is finite, we can always find an order-preserving mapping so that the resulting model will be sound and complete, relative to our observations.

In the future, event E involving objects x and y happening at time t will be denoted in the form of E(x, y, t). E(x, y, t) is an event instance. In this way, an event instance has a logical form of a predicate formula.

The predicate formula E(x,y,t) is true if and only if

1. there is a finished event that started at t

2. objects x and y are involved.

The duration of the event is denoted as |E(x, y, t)|. In this way, E(x,y,t) is viewed logically as a term which is the value of function E taking three arguments x, y and t.

[4]See the next chapter for a more detailed account of our discrete time system.

As many philosophers have argued, an event can assume both predicate and term as its logic forms [Davidson 1980]. So we may choose the forms interchangeably. E.g., we can have a function Participants which takes an event as argument. Participants(E)=u,v and E(u,v) both mean that u and v are the participants of the event E (but E(u,v) may have other participants).

In the future, we will introduce the notion of event class. An event class E can be formally viewed as an operator. When it is applied to a particular value t (time) an event instance E(t) will result.

The notion of event class is very useful in describing recurring operations that occur in a class of systems.

In what follows we will often simply write E or E(x), omitting some parameters to make the formulas more concise.

A Tentative Definition of Event

We would like to give "event" a rigorous definition. Since it is a heavily loaded word in computer science, this is particularly necessary.

We begin our discussion by recalling the example about a customer "orders a dish and eats the dish." We have pointed out that among other things, the individual "dish" being eaten is not necessarily involved in the event of "ordering."

On the other hand, it seems natural to say that the customer is an actual participant in the event. Similarly if x is married to y's father then that y in the event marryFatherOf(x,y) is also felt not to be directly involved while x is. We note that this criterion is very intuitively appealing. And, vaguely, we can see its merits: it helps in conceptual clarity and in economy of writing and computing.

However, we may want to ask what exactly it means to be directly involved.

First we point out that situation calculus [McCarthy and Hayes] does not provide an answer. In situation calculus, an event is a change of situations. A situation, in turn, is a set of relations while a relation is just a set of "tuples."

For example, since there is logical equivalence between the relations

marriedTo(x, fatherof(y))

and

marriedToFatherOf(x, y),

in situation calculus

marry (x, fatherof(y))

is completely equivalent to

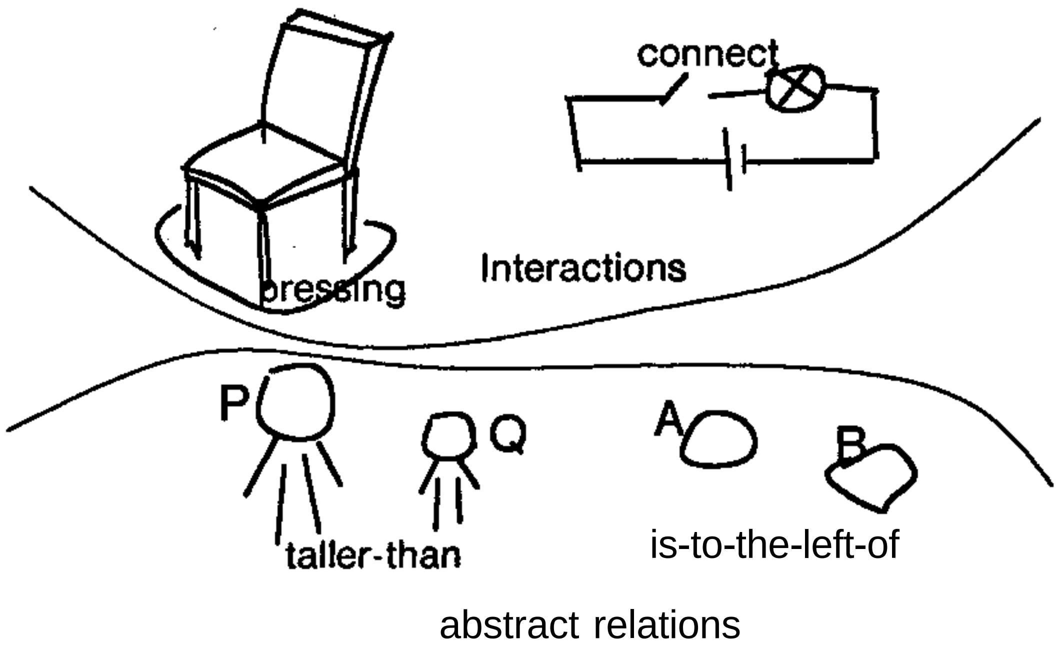


Figure 3.2: Interactions vs. Relations in General

marryFatherOf(x,y).

To be able to answer the "direct involvement" question, one has to resort to the causal relations in the observational model. We introduce the notion of interaction at the level of the observational model.

Unlike relations which can relate virtually any two things in the world (we take binary relations as an example), an interaction is defined as a relation associated with physical contact. In this way, it is a direct causal relation.

Figure 3.2 contrasts examples of relations in general with interactions.

Based on the notion of interaction, we can define physical states of objects as the combination of 1. external states, i.e., the interactions of the object with other objects, 2. internal states, i.e., the spatial relations and interactions among its immediate parts and 3. its spatial location.

A primitive physical event is defined as a state change of objects, observed over an interval of time.

Having this basic notion of event, we can start investigating how events of a non-physical nature can be defined and how a complex event can be composed from simpler ones. Before we do that, however, we have to establish a formalism that is powerful enough for our task.

That will be the topic of the next chapter.

Chapter 4

Representing Temporal Relations

We now introduce a scheme for representing temporal relations and constructing composite events. We will, in the following, concentrate on the temporality aspect of the problems and leave the other semantic or ontological aspects of events to a later time. A few temporal operators will be defined and their properties discussed.

The scheme looks like, and essentially is, a temporal logic. We are reluctant to call it temporal logic, however. While many temporal logic systems have exerted much effort in developing logic theories and proofs, we will content ourselves with a set of coherently defined concepts and notations. While some of the temporal logic systems are designed to enhance the manipulation of symbolic structures, our scheme has placed more emphasis on making the formulas easy to comprehend. As a result of this, although our notation may be succinct, the definitions are not as parsimonious as possible. Occasionally we have operators that are no longer first order although the semantics is still precisely defined.

The rationale for this is that (as is argued in the introduction) we are convinced that the first thing for the user to do is to clearly and correctly express himself. If this is accomplished, we can expect some powerful reasoning system to transform the user's formal statement to a more manipulable form and do the inference. And as long as a language is semantically well defined, this can be accomplished.

Our scheme is mainly motivated by the need for understandability. But it turns out that some of our design choices make certain manipulations easier as well. The scheme has the following three special features:

1. Representing relative orders.

   We notice that there are two kinds of temporal relations. One group of them are regular behavior patterns that are independent of a particular time. The second kind

are individual events or episodes. For the former, since it is a class of events or relations, only a relative order can be specified. For the latter, we have exact points of time for events. We consider the former more general and useful. One of our goals is to work out a scheme that can deal directly with relative orders of events. We have introduced the notion of relation class, event class, and the class of event durations, and defined temporal operators (connectives) on them for this purpose.

2. Representing events as both first order formulas and terms.

   Temporal relations can easily go to second order logic, if we talk about relations among events, but take events as predicate formulas. On the other hand, if we stay in first order logic by making an event into a term, we would not be able to make a simple statement such as event A is the necessary condition for event B. In our scheme, we have established the dual status of events. We define and use temporal connectives just like logic connectives. As a result we can build a composed event to represent complex behavior patterns and reason about them. The notations turn out to be intutive, too.

3. The use of a context dependent notation.

   The notations introduced in this chapter are intended to be used in the rest of the thesis and to make the formulas "natural" and intutive. Representing relative time instead of concrete time is a way of making things natural since this is how temporal relations are stated in natural languages. Besides this, we also use a collection of notations, including varieties of shorthand, to express the temporal relations concisely. As a result, the events, but not the temporal relations among them, get the most attention and the specifications are more readable.

   However, this also makes the notation somewhat context-dependent. The reader is forewarned that in some cases he will not be able to use simple compositional rules to find the meaning of a notation. For example, while **E** is used for the event E as a term and "E" without bold face for the event E as a predicate, the duration of **E** is represented as $|E|$ rather than $|\mathbf{E}|$. This is because that, in the context of taking a duration function, the event can only be treated as a term. For another example, in our discussion, we often allow some arguments to be omitted from the expression of a predicate formula. As to which arguments are omitted, the reader will have to consult the context to decide.

As a result the scheme provides a powerful set of constructs with which we can describe system behaviors in an concise, natural and easily understandable way.

4.1 A Uniform Treatment for Relations and Events

In our scheme we give a uniform treatment for both relations and events.

As introduced in the last chapter, an individual event will be represented in this scheme as

E(x, y, t, t').

When treated as a predicate formula, this means that it is true that an event, belonging to the event class E, beginning at t, ending at t' and involving objects x and y, happened. So E(x, y, t, t') denotes a truth value. In some discussions, when we *emphasize* that an event is not intended to be treated as a truth value we will use E in bold face **E**(..) instead.

In our scheme, we adopt a discrete time framework, where a countable set of points are distributed on a hypothetical time axis. These points are also boundaries between time quanta. The time intervals are distances between two time points. We assume both time points and time intervals are measured in numbers, probably integers. When the time scale is uniform, intervals are integral multiples of time quanta. The system is observable only at the boundaries between time intervals.

The duration of an event is the time interval between its beginning (we will use the term "start" in the future) and its end. It is also called the norm of the event denoted by |E(t)| which should obviously mean the norm of **E**(t) as we explained earlier. Its length is at least the interval of 1 time quantum[1]. Since, for a particular event, its duration is unique, we can simplify our writing to use E(t) instead of E(t,t') for an event. It is implied that

t'=t+|E(t)|

There are two reasons for us to introduce the notion of norm: 1. It fits better with our formalism, as we will see. 2. It is more essential than either of the time points (start and end) in terms of event classes and the composition of composite events.

As we pointed out, relations can also be temporal.

In fact, in this scheme the truth value of any predicate will be assigned with respect to a particular time. In general, the truth value of a relation R at time t can be represented as

R(t)

where t is the time when the truth value for the formula is evaluated and it does not matter

[1]By the way, this is also the minimum time needed to know whether the event does happen since by the happening of an event we always mean "started and finished."

what follows afterwards. A temporal relation like this is called a *simple proposition.*

Our interest in the truth value of a relation P, however, may present itself in two other ways in regard to time:

First, a relation holds from t to t\ Beginning at t' the relation stops holding.

For example, the period during which the statement "the sky is sunny" is true cannot exceed the length of a day and perhaps less (unless you are near the poles).

A relation of this kind will be referred to as a *temporal proposition.* The truth value of a temporal proposition is determined by observation during a period which is delimited by the "start" and "end" of the proposition, t and t\ A temporal proposition is represented as

p(t, f)

which is equivalent to

Vs[t<s<t' —P(s)] A -«P(t').

The duration t'—t will also be called its norm, denoted as |P|. P is bold faced when we treat the relation as a logic term.

P(t) and P(t, t') denote quite different things in the scheme, but since they will almost always be used in diflPerent places we will, not necessarily make a distinction in notations. In the future, a simple proposition will be viewed as a temporal proposition with a quantum time span. P(t) will refer to both simple and temporal propositions. Only when we want to make explicit that P is of the latter kind, will the notation P(t,t') be used.

Second, it is obvious that some relations do not change over time. These relations are called *absolute propositions.* Since it would be too messy to require that each relation always carry with it a time argument, we may want to use P instead of (Vt)P(t) etc. However, until we have a safe way to make the notation succinct, we will always require that an absolute relation be a quantified formula.

We designate the names of individual relations belonging to a relation class by the same predicate symbols. For example, P is a relation class and has P(t), P(t') and P(t, t') etc as its members. It also has P(x, y) and P(u,v) and perhaps (Vt)P(t) as subclasses. Similarly, P(x, y, t) or P(x, y , t, t') are instances of this class.

E(t), E(t') and so on form an event class.

4.2 Extending the Formalism: Predicate Classes as Operators

We will base our scheme on a first order logic with equality which, among others, has the following ordinary logic symbols with the ordinary meanings [Manna] (cf. the section on notation in the introduction):

A,V, -., ≡, V, 3, and=.

We use "—•" as the implication symbol.

We will use the following notations for common place predicates:

≥ as "greater than or equal,"

≤ as "less than or equal,"

T^ as "not equal."

If the formulas we handle are time-independent individual relations or events then all the results of the old logic remain valid for formulas constructed using these old connectives. However we need to extend the logic a little to accommodate the capability to handle event and relation classes, especially when they are time dependent.

In this section we will adopt the following naming convention:

1. Time variables will be t, t', ti,t2, etc.

2. Simple propositions will be denoted by P(t), Q(t), Pi(t) etc.

3. Temporal propositions, when they are to be distinguished, will be denoted by P(t, t'), Q(t, t') etc. Otherwise they will be denoted as P(t), Q(t) as well.

4. Relation classes will be denoted by P, Q, etc

5. Events will be denoted by E(t), e(t), or e, ej, etc.

6. Event classes will be denoted by E, Ei, etc.

7. When events and relations are not to be distinguished, they will be denoted by X(t), Y(t), Z(t), etc.

8. When event classes and relation classes are not to be distinguished, they will be denoted by X,Y, Z, etc.

In this chapter, when event classes and relation classes are not to be distinguished, we will *for convenience* refer to them as *predicate classes.* We remark that this is a sacrifice to conceptual clarity since predicate is a notion in the logic theory whereas relations and events are entities in the model.

In general, we would like to view predicate classes as operators, mathematically. For the same purposes of convenience, the class of durations of events will also be viewed as operators. By applying a predicate class to a time point t and possibly other arguments, we get a predicate formula. Similarly, a duration class applied to a time t will yield an instance of duration.

Because the temporal nature of predicate formulas, we require that these operators satisfy the following axioms:

Let t denote a time variable, u denote an instance of a type other than time (such as domain objects or pure values)[2], XX, X, Y, X(u) and Y(u) denote operators that operate on t, where XX is in the form of a formula where all the arguments except the time are instantiated, + denote the operation of addition between a time point and an interval, and h denote a function of t. Namely, if we denote the set of time (point) variables as T, the set of time intervals as L, the set of objects as U, the set of predicate classes as H, and the set of predicates as X, we will have

tGT,ueu,xxes,xes,YGS,|X|,|Y|GL

XX: T ->X;

X: U - 3 ;

X: L ~>2;

|X|: T ->L;

+: T x L -»T ;

+:L x T ->T ;

h: T ->T.

First we will have axioms as follows:

[X](t) = X(t)

[|X|](t) = |X(t)|.

These axioms reflect the fact that X is a predicate class, which is instantiated as an individual relation or event when applied to a particular time. |X| is the *norm of the class* X. The intuitive meaning of |X| is the "duration of class X," or the generalization of individual durations of events in class X.

[X(u)] (t) = X(u, t) (3.4.1)

as an axiom defines a subclass of predicates, generated from the predicate class X. If X stands for the event class eating, and u stands for person John, X(u) would stand for John's

[2]**The notion of pure values as abstract objects is explained in the chapter on domain categories.**

eating. John will eat more than once and all his eatings form a subclass of the event class eating. This class can also be instantiated.

Lastly, we introduce the axiom for delayed predicate classes.

$$X(|Y|)(h(t))=X(|Y(h(t))|+h(t)) \qquad (3.4.1a)$$

When h(t)=t, we will have

$$[X(|Y|)]\ (t) = X(|Y(t)|+ t).$$

The intent of this axiom is to handle the general class of occasions when one predicate is said to immediately follow another. The intuitive meaning of (3.4.1a) is that if an event x of class X happens immediately after event y of class Y, and event y happens at time h(t), then the time event x actually starts would be h(t) plus the duration of Y(h(t)). In other words, the starting time of X would be delayed by |Y(h(t))| for a time point h(t).

4.3 Implicit Instantiation

The reader may have noticed that a formula, say P(x,y) now may have two meanings. First if an assertion P(t) is true for all t then we may want to use just P. E.g., if the formula color(Clyde, grey, t) means that the color of the elephant Clyde is grey at a given time t then color(Clyde, grey) can mean Clyde is always grey. As the second meaning, if Clyde changes his color frequently because of disease, color(Clyde, grey) will be an operator, by applying it we get relations that hold at a particular time. We could insist on using $\Box$ [3] whenever we mean the first sense. But that would be too burdensome in the long run.

A little reflection gives us some hope. If we have an event, say "washing clothes," we may tend to find out its time to identify the event because clothes need frequent cleaning in general. However, if one says "the letter is burnt," there is no ambiguity as to which event we are talking about, because a letter presumably can only be burnt once.

But this is because we have been taking the whole past and future as the background in referring to an event. If we have properly selected the context, many events can easily establish an identity without an explicit time. In natural languages, a speaker uses his common sense and knowledge to find the context. Our scheme is formal, but we could use other events as delimiters to cut off a piece of a time stretch and make the event unique. In that case, a class of predicate will have only one instantiation and simply writing P(x,y) would implicitly refer to a single event. For example, if there is a sequence of events "write(Mary, l:letter); receive(John,l); read(John,l); burn(John,l)" and we know there is a

[3] It is mentioned in the section on notation in the introduction.

unique event "write(Mary, 1)" then the identity of John's reading is established, too.[4]

The above argument reveals that the notion of implicit instantiations provides a shortcut for connecting predicate classes and predicate instances. The old and familiar formulas like P, Q, and so on now make sense again. There could be more than one interpretation when they stand alone, but the interpretations are conceptually unified.

In the next few sections, we will be introducing operators that apply to predicate classes, i.e., secondary operators. Composed predicate classes will result from applications of these secondary operators. We define the equivalence of two *composite predicate classes* P,Q as

Let T be a time interval

$[P \equiv Q]_T \equiv VtGTP(t) \equiv Q(t)$

Thus the notion of implicit instantiation works for them as well.

Next, we will introduce logic connectives that connect event classes to each other as well as connect event classes and relation classes.

4.4 Extending the Semantics of Logic Connectives

In the extended logic, all the old logic connectives will keep their standard meaning when used on the old kinds of formulas, with or without time variables.

Now we define their meanings when used on relation or event classes,

X A Y, X V Y, and ~>X are predicate classes, just like relation class P or event class E. They are not symbols for individual events or relations but the symbols for event or relation classes. On the other hand [X A Y](t) or [X V -«Y](t') are individual events or relations as P(t) or E(t) are. This uniformity, however, does not hold for negations.

When X is an event class E, -iE should be treated as a usual relation class symbol which can have both simple and temporal propositions associated with it. This is based on the fact that -iE does not have any state changes associated with it. Treating it as an event, we will have difficulty building a coherent theoretical framework.

The problem with this is how to handle "not not X": On one hand, because the first negation has produced a relation, we have to accept "not not E" as relation since, in general, the negation of a relation remains a relation. On the other hand, we may want to think "not not E" as E itself.

It is possible to have an axiom to make "not not E" be "E" again but we now leave this open which means it is currently not true [[~«»E] = E]. But E —>-i«E if E is an event class.

[4]Of course, the principle stated for model mapping should be observed. I.e., if an instance of a class is selected for a tune interval, all the instances should be selected.

Namely, let the set of predicate classes be **X**,

$\neg$: $\mathbf{X}\to\mathbf{X}$

$\wedge$: $\mathbf{X}\times\mathbf{X}\to\mathbf{X}$

$\vee$: $\mathbf{X}\times\mathbf{X}\to\mathbf{X}$

Their meanings being

$[X \wedge Y](t) \equiv X(t) \wedge Y(t)$

$[X \vee Y](t) \equiv X(t) \vee Y(t)$

$[\neg X](t) \equiv \neg X(t)$

That is, when combined by the logic connectives, the predicates are meant to be true *simultaneously*.

As well as $\wedge$, $\vee$, and $\neg$, we have

$X \to Y\ (t) \equiv X(t) \to Y(t)$

$[X \equiv Y](t) \equiv [X(t) \equiv Y(t)]$.

It is also convenient to have

$X \oplus Y \equiv [X \wedge \neg Y] \vee [\neg X \wedge Y]$.

Now we have the following derivation, the meaning of which will be explained shortly.

$$\begin{aligned} &\neg X \vee Y(|X|)\ (t) \\ &\quad\equiv[\neg X](t) \vee [Y(|X|)]\ (t) \\ &\quad\equiv\neg X(t) \vee Y(t + [|X|](t)\ ) \\ &\quad\equiv\neg X(t) \vee Y(t+|X(t)|) \\ &\quad\equiv X(t) \to Y(t+|X(t)|) \end{aligned} \tag{3.4.2}$$

Formulas without time variables

Even when temporal factors are not the main concern of an assertion, the use of predicate classes is helpful because we need not introduce any redundant time variables or connectives just for temporal relations.

E.g., for a sentence "In the winter when the roads are wet in the valley, the roads in the mountains will be icy," considering the relations to be simultaneous, we have

$\Box$Winter $\to$[($\forall$x:road)wet(x,Valley)
$\quad\quad\to$($\forall$y:road)icy(y,Mountain)]

or alternatively

$\Box$[($\forall$x:road)wet(x, Valley) $\to$
$\quad\quad$($\forall$y:road)Icy(y, Mountain)] when Winter.

where $\Box$ and "when" are temporal operators to be introduced shortly.

Note we have reserved the following definitions for use of ($\forall$) and ($\exists$)

$$(\forall u)\ X(u)\ (t) \equiv (\forall u)\ X(u,t),$$

which means that for all $u \in U$ an event happens and all the events happen simultaneously. The existentially quantified events,

$$(\exists u)\ X(u)\ (t) \equiv (\exists u)\ X(u,t),$$

is an event abstraction.

An example use of quantified events can be "if one of the children cries, the nurse will come," which translates into the following:

$$\Box[(\exists x{:}\text{child})\text{cry}(x) \rightarrow \text{comeTo}(\text{Nurse}, x)]$$

This looks like an ordinary formula, but actually we have clearly indicated here that the nurse comes right away when the crying of a child is detected. The temporal relation is imposed by the formalism.[5]

It should be obvious that, from the way these connectives are extended, any formula that was true as $G(X(t), Y(t))$ will be true for $G(X, Y)(t)$. In other words, all the old inference rules are now applicable to formulas formed by classes connected by old connectives.

4.5 Temporal Connectives and Time Points

Next, we introduce temporal connectives.

First, we note that we will use two temporal operators widely used in most temporal logics, i.e.,

$$\Box X \equiv (\forall t{:}\text{time point})X(t)$$
(always X)

$$\Diamond X \equiv (\exists t{:}\text{time point})X(t).$$
(sometimes X)

Temporal Connectives

We introduce the following group of temporal connectives:

$$X{:}Y\ (t) \equiv X(t) \wedge (\exists t')[Y(t') \wedge 0 \leq t'-t \leq |X(t)|-|Y(t')|]$$
(X contains Y)

$$X/Y\ (t) \equiv X(t) \wedge (\exists t')\ Y(t') \wedge 0 < t'-t \wedge |X(t)| > t'-t > |X(t)|-|Y(t')|$$
(X overlaps Y and X begins earlier than Y)

$$X;Y\ (t) \equiv X(t) \wedge (\exists t')\ Y(t') \wedge t'-t \geq |X(t)|$$

[5]A more realistic formulation would be using $\underline{!}$ (see the next section), since the formula really means "simultaneous" while what we need is "immediately."

(X precedes Y)

$X \,!\, Y\ (t) \equiv X(t) \wedge (\exists t')\ Y(t') \wedge t'-t = |X(t)|$

(Y immediately follows X)

$X :;\, Y \equiv [X{:}Y] \oplus [X/Y] \oplus [X;Y]$

(X begins before Y)

$X\&Y \equiv [X{:}Y] \oplus [Y{:}X] \oplus [X/Y] \oplus [Y/X] \oplus [X;Y] \oplus [Y;X]$

(X and Y both happen or X and Y are concurrent)

It can be easily seen that

$X :;\, Y\ (t) \equiv X(t) \wedge (\exists t')\ Y(t) \wedge t \le t'$

It can be proven that

$X\&Y\ (t) \equiv X(t) \wedge (\exists t') Y(t')$

As a use of the above ordering operators, we cite the following:

```
VisitTree(a tree) ≡
  if leaf(tree) then VisitNode(tree)
      else {VisitTree(LeftTree(tree));
      VisitNode(root(tree));
      VisitTree(rightTree(tree)); }
```

This, perhaps, will not be a surprise to anyone who knows Pascal. But one should be surprised since what he sees is not a piece of a program but a specification with clearly defined semantics. If we would like the specified events to be concurrent, we can simply replace the connective ";" with "&" or even "$\wedge$," although a strict simultaneity may not be possible.

As their corresponding rules, we can have

$\mathbf{X{::}Y} \equiv X{:}Y$

$\mathbf{X//Y} \equiv X/Y$

$\mathbf{X \prec Y} \equiv X;Y$

$(\mathbf{X \succ Y} \equiv \mathbf{Y \prec X})$

$\mathbf{X \prec_! Y} \equiv X \,!\, Y$

$\mathbf{X < Y} \equiv X :;\, Y$

We note the symbols on left hand sides have been bold faced since the events here are treated as terms.[6] These are absolute propositions which state the rules that one event always contains, or overlaps, or follows another event whereas their counterparts themselves

[6]You may want to think of them as second order formulas. But this is practically more complex and not necessarily superior theoretically.

are composite events. In the future, we may not always use bold face but the reader should bear in mind this conceptual distinction.

These rules find their use in following examples. To represent "John gets up in the morning" in ordinary logic, one would say

$$(\forall t)\ \text{getUp}(\text{John}, t) \rightarrow \exists t'[\text{morning}(t') \wedge t' \leq t \leq (t' + |\text{morning}(t')|)]$$

In our notation[7] it will be

Morning:: getUp(John)

Note also, in using X::Y, the assumption is that X and Y will both happen. This assumption does hold here.

Similarly, "Leaves are green at first then yellow" is

$\Box$Green(a leaf) $\prec$ yellow(leaf)

We also define

$$X \text{ when } Y \equiv X{:}Y \wedge [[Y{::}X] \equiv X \wedge Y \wedge |X| = |Y|].$$

It can be used in cases like "John never gets C's when he is in school":

$\Box$not getGrade(John, "c") when At (John, School).

(One might represent it as Student(John) rather than At (John, School), but that is another matter).

We now introduce a notation used to abbreviate repetitive use of connectives (logical or temporal).

Assuming **op** is an connective and x_i, x_{i+1} ...x_l are classes, the "product" $\prod$ is defined as

$$\textstyle\prod_{i=k,l}(\mathbf{op})\ \mathbf{x} \equiv \mathbf{x}_k\ \mathbf{op}\ \mathbf{x}_{k+1}\ \mathbf{op}...\mathbf{x}_l$$

We then define:

$$\text{repeat for i=m to n}[X(i)]\ (t) \equiv \textstyle\prod_{i=m,n}(;)X(i)\ ] \wedge \text{start}(X(1))=t$$

$$\text{repeat n times } X \equiv \text{repeat for i=1 to n } X$$

$$\text{repeat}[X]\ (t) \equiv \exists n[\text{repeat n times } X] \equiv [(\exists n) \textstyle\prod_{i=1,n}(;)X\ ]\ (t)$$

Having defined these constructs, the correspondence between a statement in this language and one in a high level language can be immediate. For example, for Pascal code such as

```
"begin x:=0;
  while x<3 do begin x:=x+1; x:=x-1 end"
```

The specification will be

```
{x:=0 ;
      repeat [x:=x+1 ; x:=x-1 ] when x <3 }.
```

[7]It is hoped the intuition that getting up happens during the morning is better expressed in this way but this is not the claim.

Let us warn the reader, however, that here the Pascal code can no longer be viewed as instructions, but should rather be taken as phenomena we would observe inside the computer at a certain level of abstraction.

Assuming that U is a finite set, u is both an arbitrary element in this set and an argument in class X but not a time, also assuming that m is the cardinality of U, i.e., m=cardinality(U),

$(\forall u)(\&)\ X(u)\ (t) \equiv$

$(\forall u)(\exists t)X(u,t) \land Min_u(\text{start}(X(u)))= t \equiv$

$\prod_{i=1,m}(\&)X(u_i)\ ] \land Min_i(\text{start}(X(u_i)))= t$

(X(u$_i$) happen concurrently and in no particular order)

Further assuming that there is an agreement on an order in U, e.g., $\prec_U$, we should have

$(\forall u, v \text{ in } U)(u=v \lor u \prec_U v \lor v \prec_U u),$

then, if $(\forall i{:}\text{integer})(u_i \prec_U u_{i+1})$, we define

$(\forall u);\ X(u)\ (t) \equiv [\prod_{i=1,m}(;)\ X(u_i)\ ]\ (t)$

Some formal properties of these connectives or operators will be given in the appendix.

Special Time Points

In this theory, the notion of time is mostly relative. But a few important time points are worth mentioning:

Current time, will be always denoted by "T";

The beginning of the system, will be considered t=0;

"infinity" is the other end of the spectrum.

Based on the notion of current time, we define "in the past" as operator "#":

$\#X \equiv (\exists t)\ X(t) \land t<T$

(In the past, some time X)

where T is the current time.

The past operator finds wide use. For example, if one believes "versatile experience" means one has been in business, academic and military and so on, he can define "Versatile-Experience" as

VersatileExperience(a person) ≡

#[person = an officer in the military &

person = an executive in a company &

person = an actor in show business &

person = a professor at a college].

The "Past" operator is typical in limiting the range of the values of the time variable and in so doing, implicitly instantiates an event class. For example, we may translate "after he arrived, John lived in the dorms" into

$\#(\exists x{:}\text{room}, y{:}\text{place})[x\in \text{dorm} \wedge \text{stay}(\text{John}, x)\wedge$
$\quad \text{stay}(\text{John}, x) \succ_{!} \text{arrive}(\text{John},y)\ ]$

But # does not always make events unique. The defining of buffer contents is a good example. If we think "the contents of a buffer is all the objects that entered but did not leave the buffer," then we will have

buffer.contents = {x: object |
 #[[repeat[enter(x,buffer);leave(x, buffer)]];
 enter(x,buffer)!¬leave(x, buffer)]

In this formulation we have also shown how the negation of events can be a useful relation.

We can generalize # to temporal relations within an arbitrary interval. That is, we replace t=0 to $t=t_1$and t=T (current time) to $t=t_2$. This will, as we pointed out earlier, generates a time stretch, helpful in identifying events. Formally

$X \text{ between } (t_1{:}\text{time}, t_2{:}\text{time})\equiv\exists t[t_1<t<t_2\wedge X(t)]$

We introduce two "constant classes," truth and falsity, in the end, such that

$\Box[\text{truth}=\text{true}]$

or equivalently

$(\forall t)\text{truth}(t)=\text{true},$

$\Box[\text{falsity} \equiv \text{false}]$

or equivalently

$(\forall t)\text{falsity}(t)=\text{false}.$

Still, representing some temporal relations may be tricky; for example, "A or after a while B." Theoretically, we can write something like

A ∨ [not B ; B] or alternatively

A ∨ [not A ; A]

Here the relation class *delay* may be useful. It is defined as

$[X!\text{delay}!\ Y](t) \equiv X(t)\wedge Y(t+|X|+|\text{delay}|).$

Now we can just say

A ∨ [Delay;B].

When the delay is a fixed length, we can use the basic property of the class notion, for example: "A or an hour later B" could be

$A \vee [\text{delay}!B]\wedge |\text{delay}|=1.$

We conclude this section with an example combining time point representation and relative ordering.

As we have mentioned, any predicate in the system can have a time argument. Usually this time argument need not be present. When it is, that is the time at which the predicate becomes true. An event or temporal proposition can have two arguments of type time indicating the beginning and end of the predicate. All of these time values are counted with the origin being t=0, unless otherwise stated.

Using this scheme, we represent the sentence

"the system crashed 3 times last week. The longest was on Monday which lasted two hours." as

repeat 3 times [crash(System)] between (t_1, t_2)

and assuming quantum time be hours,

$\max_{t_1<t<t_2}$ (|crash(system,t)|)=
|crash(system) between (t_3, t_3+24)| =2

where

t_1=SundayDate(LastWeek()),

t_2= SaturdayDate(LastWeek()),

and

t_3= MondayDate(LastWeek()).

The functions LastWeek, SundayDate etc are defined as

```
LastWeek() =
  T -
  NumberOfDayInCurrentWeek(DayOfWeek(T))*24
      -timeOfDay(T) -7*24
SundayDate(t) =t
SaturdayDate(t) =t+6*24.
```

Event Sequence and Selector Functions

Any actual complex event instance is a sequence of events.[8] Their generalizations at the highest level may be an event class containing concurrent events which makes the event class difficult to handle. At an appropriate level, we may have a class of sequential events.

[8]There are two related senses of "sequences of events." One is the mathematical sequence, i.e., the events as they are described textually. Another is the temporal sequencing of the actual events. Here it is necessary to note that by "events" we mean that no simple propositions are counted because the sequence as a description cannot be unique then.

If this event class is E, then the sequence of events (classes) it corresponds to is denoted by **||E||**.

This comes from the need to make use of the concept of sequence as a way to refer to a particular component event in the whole composed event. For ease of reference, two selector functions are defined.

elt(E, Ei) will refer to the only $E_{t'}$ in E. If there is more than one $E_{t'}$ in E or there is no $E_{t'}$ in E, the function is undefined.

elt(E, n, E,-) refers to the n-th occurrence of E_{t-} in E. If there are fewer than n E_{t}'s in E, the function is undefined.

The order of the occurrence of events is the temporal order in which they occur. Since the same event cannot happen to the same set of objects at exactly the same time, no ambiguity will result.

Generalizing "If Then"

The last connective we introduce is a generalization of the logic connective "if then." This generalization has very interesting implications, as we shall see.

We define the following connectives "=>" or "if then," "«=" or "Onlylf then," and "<*" or "Iff then" as follows:

X =>Y (t)≡

X(t)-Y(t+|X(t)|)s

• X-Y(|X|) (t) s • •

HX(t)VY(t+|X(t)|)

if X then Y (t) ≡ X =>Y (t)

(if X then afterwards Y)

X <=Y (t) ≡ Onlylf X then Y (t) =

Y(|X|) →X ≡

Y(t+|X(t)|)-X(t)

(Only if X then afterwards Y)

X *&Y* (t) ≡ Iff X then Y (t) =

If X then Y (t) A Onlylf X then Y (t)

Obviously, this definition of "if then" will still hold when the time span (or norm) of X is 0, that is, when X and Y are simultaneous. So it indeed generalizes the "if then" in unextended logic.

This generalization is essential. To see this, let us consider our everyday notion of "if then."

Consider the following sentences:

1. **If** the engine is too **cold,**
   the **engine is** warmed **up** one **more minute.**
   In winter, **in** the East,
   if there is waterfall then it is snow.
2. If the car started I run.
   If the light becomes green then the car moves.
3. If a car is started, its engine is warm.
   If father(John, Joe) then Parent(John, Joe)
4. If the car engine is started, the car moves.
   If the engine is too cold, the car stalls
   If the car has started, the crowd moves.

Here we have four different uses of "if then."

1. The first one is the one expressed by the logic connective "imply" in our extended sense. If not checked against arbitrary time, it holds for the ordinary sense of the connective as well. We note conditional statements in programming belong to this class, if the statements are viewed as narrations rather than imperatives.

2. The second class expresses both material implication and temporal ordering, i.e., "if X then afterwards Y." X is always an event. Demons in some AI languages or the "waitfor" construct in some simulation languages belong to this class.

3. The third one is a generalized version of class 1. It is the one captured in logic by "X implies Y," since in most case we mean "it is always true that X implies Y."

4. The fourth is a generalized version of 2. I.e., "it is always true that first X then afterwards Y." This expresses what we may call causal laws, since the relation is temporally ordered and regular.

Now that we have introduced "=>," they are correspondingly represented in our formalism as

1. X -»Y
2. X =>Y
3. D[X -*Y]
4. D[X =>Y].

In the terminology of the AI knowledge base, the first two cases are facts, but the latter ones are rules.

It is important to note that, for the sake of modeling actual systems, the newly introduced connective "=>" can only be a rough approximation. As a matter of fact, it is often the case that an event caused by another event will start before the first one finishes, but if the time interval is not too great, we take them to be in this relation.

Now we show how this "if then^{51} can be used. The example is a statement: ^{u}In the winter if it rains during the day, it will be icy at night"

As we explained earlier, there are a few predicates whose common meanings make them obvious temporal propositions. Since the predicates Day and Night become true alternatively each day within 24 hours, they are temporal propositions.

If we do not want to use the new connective, the statement will be put in a formula

(Vt) Winter(t) ->
[Rain(t) A Day(t) -•
(Vt')[0<t'-t<24 A Night(t') ->icy(t')]]

In our language, we will have

DWinter —>[Rain A Day =>Icy when Night]

which, noting

X when Y ≡ Y:X A X:Y,

can be easily expanded to

(Vt) Winter(t) ->
[[Rain A Day](t) -+
(Vt')[|Day(t)|+t]<t'<
[|Day(t)|+t+|Night(t')|] -
[Icy(t') A Night(t')]]

If one likes, the following axioms can be used to define Day and Night:

DRise(Sun) =>Day
DSet(Sun) =>Night
DDay ≡ not Night.

It is interesting to consider how to represent the sentence

"If the guide comes and the weather is fine, the tourists will go"

[Come(Guide);Fine(Weather)] =>Go(Tourists)

This tells us that the weather need not to be fine for the Guide to come but it does need to be fine right before the tourists go.[9]

[9]**This may be a causal relation, but it involves human decision making.**

Chapter 5

Causation and Interaction

5.1 The Notion of Causation in Philosophical Enquiries

The quest for the nature of causation started in the agè of Aristotle [Bunge][1]. Hume, in his *Treatise* and *Enquiry*, first related causation to fundamental problems in philosophy and gave a critical analysis of the concept in ordinary causal talk. Subsequently Mill pushed the discussion to a greater depth by explicitly analyzing various aspects of causation in great length. ([Mackie]) After surviving a massive attack by positivism, which claims that causation is a "myth" and that the only reason causation is allowed to exist is because it "like the monarchy, is erroneously supposed to do no harm," the problem of causation is getting increasing attention and is now a central theme in metaphysics. Various theories have been formed to account for causality and causal thinking. Many of these theories have gone into very technical details which is a sign of maturity in scientific inquiry.

The reason for this deep and prolonged enthusiasm is simple.

Causal phenomena are a constant factor in our everyday lives. Determinicity, a generalization of causality, is the cornerstone for any scientific investigation. As Hume put it, causation is "to us the cement of universe."

Causality is of interest to philosophy for its own sake too. This is because understanding causation plays a key role in understanding major philosophical problems such as free will, action, agency, time, events, natural laws and empirical knowledge. There is another reason why the research on causation in philosophy is particularly interesting to us: The analysis of causation has gone so deep that it is conducted in a more technical fashion than in some other topics in ontology, e.g., events and objects. Consequently various formal treatments

[1] The single important source of literature for an investigation of our sort is from the works in philosophical enquires. This proves particularly true of the notion of causation. This is an evidence of the central place of this problem

of the above concepts can find a testing ground in the formalization of causation.

For our purpose, the discussion on causation can be viewed as going along two lines. The first line is the debate on the metaphysical nature of causation. The second is a technically detailed analysis, i.e., formalizations of the concept.

The problems in the first line can be simply put as follows.

1. Is there such a thing as causation "in the objects" (a la Hume)?

   Isn't what we used to call causation really regular succession of unrelated phenomena? Or is it just our own free association of our impressions (sensations)? The problem concerns itself with the objectivity of causation and often the objectivity of our knowledge in general.

2. Is the conceptualization of causal relation interesting and worthwhile?

   Does the notion of functional dependencies (or even differential equations) completely subsume causation [Russell]. Also, do the new developments in human knowledge, particularly modern physics, reject the basic notion of cause and effect? In another words, is the notion of causation obsolete?

   Facing these problems is an essential prerequisite for any discussion on the issue. However, they are reflective in nature, often presented in the form of speculative arguments and do not lend themselves to easy formal treatment. We, in what follows, will not indulge in any discussion in this direction. We will just take it as our basic assumption that objective causation does exist and conduct the analysis on what this causation should be like.

Next, we briefly review various aspects of problems in the second line. But a serious discussion of the issues will appear only in the next subsection when we present and justify our own definition of causation.

The issues along the second line are roughly as follows:

1. Is it possible to give a complete characterization of causation in logic?

   That is, is it possible to specify causation completely in terms of logic formulas (presumably introducing some special variables, e.g. for space and time)?

   There seems to be a consensus *among a considerable number of philosophers* that it is not. An argument in principle is that if causation is more than regular succession, it should be because there is something behind what is *directly* observable in space and time.

In practice, every logic definition trying to completely characterize causation has been shown to fail [Brand]. For example, for the view maintaining that a cause C is a logically sufficient condition for the effect E, it is easy to see that if C is the cause then for any clause P, $C \wedge P$ is the cause of E.

In a contemporary formulation, it is commonly accepted that a causal statement can be interpreted as a *counterfactual statement*. For example, "X caused Y" would be meaning "had X not happened, Y would not have happened." Note that ordinary first order logic is not good for this formulation because the formulas therein are supposed to be factual (although the formalism of logic by itself should be neutral, but the model for a factual statement just can not be the same as a counterfactual one). There is a proposal for a logic with "subjunctive modal operator," but substantial work remains to be done.

It might be possible to develop a modal logic with *a causal operator*. But the present attempt made by Arther Burk is not a successful one. In his definition, if

$X \Rightarrow Y$ ($\Rightarrow$: causal operator)

then

$\neg Y \Rightarrow \neg X$,

which is too far away from our ordinary sense of causality and rejected by most authors [Brand].

2. Does causation entail some genetic or productive power?

   If the answer to problem 1 is "no," then the answer here should be "yes." In physical systems the genetic power is embodied by the concept of force. In general, it requires that a cause is associated with an object that exists and acts. This is called "genetic principle" [Bunge].

3. What is the logical implication of causation?

   I.e., if we can not completely characterize causation in logic, how can we capture the logical aspects of the notion.

   Almost all possible combinations have been tried. A majority of philophers support the theory of "necessary connection," which says that the cause necessitates the effect [Sosa] [Brand]. This implies that the cause is a *logical (material) sufficient condition* of the effect.

This view, if taken literally, is unrealistic because we can never enumerate the sufficient conditions for anything. Striking a match causes it to light, but oxygen need be present, the match should be dry and so on. Failing any condition, the match will not be lit. Therefore this view is always complemented by so called *"ceteris paribus"* clause. That is, everything else being equal, if a condition is enough for something to happen or not happen then it is the cause.

But still this solution can be challenged. For example, in the statement "the short circuit caused the fire in the house," presumably there are other similar critical factors present at same time. For example, the fuse was not right, insulation for the wires were not appropriate. To alleviate this problem and eliminate the problem of *ceteris paribus* clause, Mackie suggested that a cause should be *at least* an insufficient but necessary part of an unnecessary but sufficient condition (the famous INUS condition) [Mackie].

In his formulation, if

ABC or DEF or GHI -+P

then A, or ABC, or ABC V DEF are all considered the causes of P.

Mackie's formulation, however, was not clear about what kind of things A, B, C etc are. If we take them as predicate formulas in general and "or" as meaning logic connective V then we would have a paradox.

For example, if

A V B ^ P

then A and B are the causes of P. However,

A V B ≡AV (-.A A B).

Does this mean -iA is also the cause of P?

4. What is the distinction between a cause and a condition?

The problem also has a terminological side. That is, where is the difference between what we usually think of as "a cause" and what we think of as "a condition."

Undoubtedly, those maintaining a cause is a necessary and sufficient condition would not have any problem here. But for others, they have to make a choice for each condition as for what title they are to give it.

[Collingwood] suggested that in everyday conversation, we usually take something abnormal, wrong, irregular, presence of action, etc. as causes while we take their opposite, i.e., regular, normal, right and absence of action as conditions. For example, the presence of oxygen is not thought of as the cause of the lighting of the match, but its absence can be the cause of someone's suffocating to death.

This suggests that we can take the negation of a cause as a condition and vice versa since it is often true that if P is normal then $\neg$P is abnormal. This would match the naive notion of causation quite well.

However, intuitions on causation differ. As [Collingwood] analyzed, historically three senses of causation developed. At first, the notion of causation was associated with human actions causing events to human. Later both cause and effect was extended to include non-human actions. Finally a *scientific sense* of causation was developed for which the distinction between cause and condition was not a substantive issue. In fact, in [Bunge] and in AI literature only the scientific sense of causation is used.[2]

5. What is the nature of the argument in causal statements?

The first layer of this problem is ontological. The question is what can be a cause. It is generally agreed that an event can be a cause, e.g. a gun is fired, a match is struck or a circuit is short-circuited. But could a sustained state be a cause, e.g. could "the pulling of the locomotive cause the caboose to move"? If we accept events and states as candidates for causes then what about absence of events and states? For example, can the non-existence of oxygen in the circumstance of striking a match be considered a cause for the match not to light?

The second layer of problem five is more in terms of semantics.

In the examples of point three where we introduced the notion of INUS condition, the symbols A, or B, or C can be taken to mean "the fire broke out in the house" or its nominalization "the fire in the house," similarly, "the lighting of the match" and "striking of the match" versus "the match is lit" and "the match is struck."

The sentential form is more natural and its nature better understood. Also there is no particular difficulty for "cause" to relate two statements in general. Davidson [Davidson 1967] suggested that we can say "the fact that X caused it to be the case that Y" where X and Y are two sentences. More importantly, if a sentence is translated into a logic formula it would be easy to put the logical aspects of causation to work within first order logic.

[2]The notion of "commonsense" causality in AI literature is not the same as that of [Bunge]. But, in any event, it is complete different from the other two senses of causation mentioned here.

But treating the arguments in a causal statement as entities like objects, that is to say accepting the nominalized form, appears more in line with the traditions of ontology. For example, Davidson argued that events are particulars, which means an event is considered an object with identity.

6. Is the causal relation asymmetrical?

   Although the general consensus is yes, there is still the problem of determining exactly the basis for this asymmetry.

   Some maintain that a cause event has some compelling power to force the effect to exist. In fact, the notion of force in mechanics is perceived as performing exactly this role. But there are others who dismiss this view. For them the asymmetry is only in terms of temporal sequencing, i.e., the cause temporally precedes the effect. But the claim of the latter has difficulty explaining so called simultaneous causation, e.g. the pulling of locomotive causing the movement of the caboose [Taylor Richard]. This is actually another issue, to wit whether causal relation entails temporal priority.

7. Does a singular causal statement entail a law?

   For example, a statement like "Brutus's stabbing Caesar caused Caesar's death" is a causal statement. Does this statement describe a law (or instance of a law) or not?

   A statement like this can not be a law by itself. This is simply because a law is concerned with recurring behavior but the stabbing of Caesar can happen only once. Hume and some other theoreticians introduced the clause "similar circumstances" to make the "law view" work. But there is a problem of how "similarity" is defined. As a matter of fact, the murder scene in "Julius Caeser" is very similar to the real stabbing scene, much more similar than a dark New York alley would be, but nobody in the cast died. [Ducasse]. In another words, contrary to common conception, not all causal relations are lawful relations.

   It is useful to comment that lawful relations need not be causal, either. For example, statistical relations are lawful but not causal.

   But as [Davidson 1967] successfully argued, a singular causal statement entails that *there is* a law. In the case of science, what we care about are almost always lawful causal relations.

8. Does causation require temporal and spatial contiguity?

In his classical analysis, Mill give a detailed description of the principle of contiguity in causation. There is no serious challenge to this view although in terms of spatial contiguity no further corroboration exists either.

The only exception is in [Bunge], which interprets spatial contiguity as only referring to directly observable objects and dismisses it as "inconsistent" with field theory. However, if our causal talk is confined to only directly observable things, we can not talk about almost any causation in the area of social relations. On the other hand, we all know that atoms and molecules which make up what we ordinarily see are not directly observable either.

5.2 The Operational Notion of Causation

5.2.1 A Methodological Note

Before we delve into the technical discussion of what an operational notion of causation would look like, a methodological note is in order. We should make clear what we are doing here. We need to answer the questions: Are we doing philosophy, or logic or engineering? If we are doing engineering, then what are these metaphysical discussions for? What is the criterion for a good logical characterization of an ontological concept?

The answer to the first question is that we are doing engineering. Whatever the philosophical issue may be, we approach it with technical precision. The merits of our work is judged by the kind of problems it solves or analyzes. For this purpose, we often do not take the philosophical problem in its full generality. Instead, we narrow the domain so that the conceptualizations can be formalized and conflicting views can find some common denominator. Another characteristic of this approach is that we do not intend to do cognitive modeling. Concepts like object, event, and causation mean many different things to different people. We find a workable approximation and precisely define it. What we mean by a term is then what the definition means. The concept becomes an engineering tool.

But why do we need any philosophy within our concrete engineering theories? (as opposed to our concern with research methodology) The answer is that these concepts can help us approach the problems. Shortly we will examine a notion of causation developed by De Kleer et al. For them the notion can be used in producing prediction and explanation of system behaviors. Many techniques are developed based on this notion. In the actual programs, there are heuristic rules and algorithms, but no "causation" per se. However, can one then say that the work is just, say, a modified or improved constraint propagation technique, or a trick to handle problems in simulation? Definitely not. If a philosophical notion has become the organizing theme of the problem solving scheme; if the scheme appeals to users by this notion, then it *is* this philosophical notion that counts.

While our engineering technique benefits from philosophy, it offers something in return. By restricting the domain and subject of the philosophical view to logic and other modes of exact reasoning, these views can be tested and developed.

In this way, we view our work as *an applied ontology.*

5.2.2 An Operational Characterization of Causation

A causational account is crucial in constructing understandable and analyzable system specifications. Our characterization will be guided by this engineering need. We will mention how technical considerations affect our notion of operational causation when we introduce each property of causation. A later section will be dedicated to a general discussion of the role of causation in our description and analysis schemes. We will then use our notion of causation to characterize the difference between behavioral and functional descriptions. We occasionally engage in certain issues that seem to be not closely related to the particular need of specifications and descriptions. In particular, we will discuss the bidirectionality of interaction and unidirectioiiality of causation. These two notions will turn out to mark a fundamental difference between the "common sense causality" and a scientific notion of causation and the difference between two kinds of approaches towards understandable descriptions are based on them. Some subtle issues are discussed in footnotes.

As we pointed out earlier, unless we introduce modal operators (possibly along with the mechanism of possible worlds) there can not be a definition of causation completely in terms of logic [Brand]. In particular, we do not have

[(P->Q)-(P=»Q)J

where =>is the causation operator. The best we can do is to define the logical implication of causation.

A causational [3] relation "P =>Q," where P and Q are events or relations, implies:

1. A material implication: [P=>Q]—>P —>Q.

   For example, hit(Car , Mary) =>sad(friendsOf(Mary)) logically implies that Mary's being hit by a car is a sufficient condition for her friends to be sad.

   Choosing material implication means that we have, in principle, accepted the "sufficient condition" view of causation. The reasons are as follows:

[3]We will avoid defining "causal" relation but will use the word "causationar instead. This is because "causal relation" is often used to indicate that a relation can be a cause or can be causally significant. There is a direct parallel with the way the term "temporal relation" is used.

First, this view is the view held by the majority of philosophers and reconciles with the ordinary sense of causation well enough.

Secondly, it makes notation and implementation simple. For example, by this axiom, the post-condition of an event, i.e., the changed situation after the event has happened[4], if it is a causal relation itself, would have that event as its "cause." It may then be part of the cause for another event. A chain of implications among events and causal relations would form a chain that is conceptually easy to grasp[5] and computationally easy to implement.

The INUS condition view is more appealing when it is taken as a philosophical argument. But in the first place, it is too complex. If we adopted that view, each time we talk about the happening of events we would have a dozen causes. Secondly, and more importantly, we have established that we are not talking about reality with an infinite number of possibilities. We are dealing with reality through a "filter" of a so called abstraction mechanism. The consideration of causation has already sneaked in, to some extent, in deciding the relevance of phenomena. Given that there would be only *a limited number of prior conditions for a supposed effect, as the pool for selecting causes*, what is left is for one to decide which set of conditions is inaugurated with this title. The practical difference here is less significant than when one is searching for all relevant observations.[6]

2. A temporal relation: $[P \Rightarrow Q] \rightarrow P \preceq Q$.

   Here P is not only preceding its effect Q but can also be simultaneous with Q. In the literature this is referred to as "existential priority" as opposed to "temporal priority."

   The reasons we did not use $\prec$ but accepted $\preceq$ instead is due to practial considerations of producing consistent specifications.

   The first reason is to handle *causal statements on composite events.*[7] For example, in specifying "crossing the English Channel caused him to catch cold," we have two

[4] "Post-condition" or "consequence" will be explained in more detail in the section on interactions and the section on event abstraction.

[5] This is close to usual notion of "a causal chain."

[6] We note that our choice can not be of principle importance. As long as we accept that the happening of an event requires all the relevant factors work together, it does not matter how we put the labels. In particular, when specifying the behavior of systems, the distinction between cause and condition would not be necessary.

[7] To simplify the notation we have allowed a slightly different use of $P \preceq Q$ in the context of the discusion of causation. Whenever P starts earlier than or simultaneous with Q, $P \preceq Q$ is true. This makes it easier for us to include complex composite events in our discussions.

composite events each with a long duration. If we are not talking about the appearance of symptoms but are interested in what happened in the swimmer's body, the two events seem to have happened at the same time, or at least overlap a great deal. In this case, we can only use "$P \preceq Q$" to express the situation.

The second reason is to handle absence of change (unchange). For example, we may want to say "in the hurricane the roof was not torn down, – a bolt fastened tightly caused it staying there." Here the cause and the effect are simultaneous because "nothing happened." We note that the case of "unchange" is very frequent in our specifications. Some times our goal may be an "unchange." For example the safety valve for a boiler is to prevent the boiler from exploding.

But simultaneity adds difficulty in specification checking. If we did not have this "equal" part, we would be able to know that R can not be the cause of E if it is later than or simultaneous with E.

We point out that *for two atomic events*–primitive events in the observational model[8] – (or states), only ***strict precedence holds.***[9]

This is the result of our abstraction mechanism. By definition, the consequence of a primitive event is assumed to be fully determined by the state at the time it starts. In the crossing the Channel and catching cold example, the corresponding component events are muscle movements of the swimmer and steps of his blood circulation or breathing. But each muscle movement can not have any effect until the movement is

[8]Our abstraction mechanism allows us to take complex events as primitive events in an abstractional model. See the section on event abstraction in the next chapter

[9]There are arguments advocating so called "*simultaneous causation*," which seems to offer another justification for existential priority. The classical example was Taylor's locomotive pulling a caboose. Based on Newton's laws of motion, – the reaction exists at the same time as the action, it is argued that the causation here is simultaneous.

There are both a confusion and a fallacy in this and similar arguments. The confusion is that the pulling by the locomotive and the pulling by the caboose are both causally significant, (as a matter of fact, they are interactions) and *they* are simultaneous, but the pulling of one and the movement of the other stand in a completely different relation. The fallacy here is to pass a judgement about locomotive and caboose based on the experience that the former would pull the latter, which is not always true. It is possible that the train is going backward (say, downhill) while the locomotive is pulling. The pulling then can not be the cause of movement. Even in the most simple case, when the locomotive is pulling the caboose to overcome friction on the rails and the train moves at a uniform speed, the pulling is not the simultaneous cause of the movement of the caboose. This is because if the pulling is stopped at the moment, physics tells us that the train will travel at the same speed for that very moment and an infinitesimal time unit thereafter. The reason is simple. In this case, the visible change, speed, is a first order derivative, but the cause, force, is related to the second derivative, accelaration. There is no way, theoretically, for a "simultaneous" causation to exist. The change you observe at *this* moment is caused by the force at *another* moment, although the force may remain unchanged.

performed. As a matter of fact we may be unable to observe, not to say to describe, what happened *during* one contraction of muscle.

This property can be directly used in simulating a specification. If it is specified that two events stand in some causal relation then normally this entails an implication. But if they appear in the same concurrent event, the checking program may stop and ask questions. Of course, if the time order is reversed then no implication is possible.

3. Temporal contiguity

   [P=>Q]->[[P^Q] v 3R [P^tR A R^tQ A P=*R A R=>Q]],

where P, Q, and R are all events involving interaction changes (cf. the sections on relations and events in Chapter 3)[10]. Namely, if P is the cause of Q then either P immediately precedes Q or there is another relation R that lies in between such that R immediately succeeds P and immediately precedes Q.

Although this point is least challenged in most specifications of causation, some remarks are to be made.

First, the notion -<! is based on our discrete structure of time. Therefore this contiguity does not imply that the time dimension is densely populated by events.

Secondly, for physical events or states, this contiguity, and the resulting continuity (not in the continuous mathematics sense), is obvious. However, social events involve human agents who have mental states. For example, between agent A's reading a note and doing something there may be a long time interval. To conform to this principle, we need to assume a mental state bridging our observable behaviors. But mental states are not observable.[11]

Because of the bridging effect of mental states, this property can only be used for the checking of physical events. We can always require that there is no gap in terms of time span for happening of events.

4. Spatial contiguity:

   $[P \Rightarrow Q] \rightarrow$

[10]Events that are purely location changes, e.g., something moving from A to B based on inertia, do not involve interaction changes. In a sense, the movements of this kind are better conceived as states rather than evonts. But we still treat them as events because spatial relation changes are very important for our framework, which is based on direct observability.

[11] Fortunately, that is basically the extent of the harm "mental state'[1] can do to us. In our operational model, mental states are never the initial cause or the sole cause (sufficient condition) for an event. An agent is always responding to some change in the environment. As a matter of fact, a specification is meant to describe how one's behavior is constrained by the events which have occured to him.

{[connected(P,Q)]V' '

3R[P=»R A R=*Q A

connected(P,Q) A connected(R,Q)]}

where R is a relation or event, and connected(P,Q) is just a shorthand for a first order formula:

connected(P,Q)=3u,v,w[P(u,v)A Q(v,w)A (P=P(u,v))A (Q=Q(v,w))]

where P(u,v) and Q(v,w) are *interactions* as defined in next subsection.

The reader should notice the obvious symmetry between spatial and temporal contiguities.

There is something worth noting in the formulation of both this property and temporal contiguity. That is, we did not say that

"3R [P^tR A R≺ Q...]"

but instead simply

"3R [P-dR A R≺!Q...]."[12]

The checking power of this property is obvious. For any physical object we require its presence at the place where the event happens. Of course, an event may involve a large space, for example the launching of spaceships. However, in many cases the place is fairly confined. In a restaurant the chefs are in the kitchen, dishes are cooked in the kitchen and one has to go to the kitchen to get dishes. Therefore this property can serve as a good heuristic in checking specifications.

5. Distinctness

It is very difficult to give a formulation of this property. If the cause and effect are temporally separate, they are automatically distinct. However, if we have "simultaneous causation," the distinctness is not easily captured by logic. If the effect is brought about by its only cause, then logically they are mutually necessary and sufficient, i.e., equivalent, but ontologically they are distinct.

[12]**Iii** the case of temporal relation, this says that the causal chain has no gap in terms of time. In the case of spatial relation, this plays the same role. But one should not infer from this that there is one **and** only one object connecting the objects in P and Q. What this requires is that one should be able to find the (possibly composite) connecting object by our abstraction mechanism.

One consequence of this distinctness is that the arguments of the cause are possibly different from the arguments in the effect. But this is only contingent. To use it in checking rules, restrictions have to be added.[13]

6. Externality

   -,[arg(P)=arg(Q)={x}]

   This axiom eliminates self-causation, i.e., no object can do anything to cause something to happen to itself. It is the same as

   VP,Q,x -i[P(x)=>Q(x)]

   assuming P and Q *only involves x.*

   This axiom is by itself a checking rule. If we find that an event has only one participant then something must be wrong: It has to connect to something other than itself.

7. Geneticity

   There should an axiom forbidding objects to come from nothing and it has to exist to be part of a cause. This reflects the genetic principle of causation. Its exact form and its use in checking will appear in the chapter on system models.

8. Lawfulness

   Up until now, we have been a little vague about what P and Q are. Are they required to be event classes or instances? (Remember we remarked before that the use of the word "event" might mean both.) According to philosophers they can be both but for different reasons.

   This is a point where philosophical thinking and engineering needs differ most conspicuously. As we have seen from the preceding subsection, the original sense of causation comes only from singular causal statements and the law it implicitly entails is secondary and not directly identifiable. In the broad context where philosophical enquiries are conducted, classifications are the ending points rather than the starting points.

   In our scheme, however, we have a set of semantic bases established through model mapping at the very outset. The existence and distinctions among classes of objects and relations are ready at hand. The problem is to provide a framework to describe

[13] **Conceptually this is not even quite correct since the distinctness is about the events not the involved objects. We will explain it in the section on bidirectionality of interactions.**

them. The causational structure is not only assumed to be known but is used in the very process of setting up appropriate mapping between models.

Simply put, when describing systems we are knowledgable enough (which is the case of writing specifications), we have got over the stage of individual observations and are in the position to make general statements. On the other hand, only a statement with sufficient generality can be of significant use. And these justify our decision to seriously investigate only causational relations between event or state classes.

We define *immediate causation* for the convenience of discussion.

P immediately causes Q if $P \Rightarrow Q$ and $P \prec_{!} Q$.

There are two corollaries:

1. connected(P,Q)
2. Not both P and Q are relations.

5.2.3 Interaction as Physical Causal Relation

It was suggested above that the cause has some "power" which makes an effect happen or come to exist. This naturally leads us to look into the causes themselves. Interactions among physical objects are most conspicuous in this regard.

As we outlined earlier, interactions are causally significant relations in the sense that they *"cause"* events to happen. Recalling the causal statements we encountered, the pulling of a caboose by a locomotive or contact of a match against some surface are both interactions. The pulling lasts an extended period of time while the contact is only momentary. The pulling can be viewed as the direct cause while the contact is part of a complex cause.

Being of similar nature as "causation", interaction is also not subject to complete logical characterization. In fact, we should say causation has "inherited" the elusiveness of interaction since the latter is really the basis for the former. The kinds of interactions differ depending on the problem areas. Typically they are physical forces or forces in a metaphorical sense. E.g., for oxygen and hydrogen to react, we may assume the so called chemical affinity (to stay at a level sufficiently abstract and appropriate).

As for its formalization, we follow the same procedure as for causation and introduce the following axioms. However, we note that unlike causation the notion of interaction plays a role mainly in the observational model and in the mapping. We can hardly identify interactions at the level of abstractional model. Therefore we can not find direct use for the axioms. On the other hand, this makes the concept of interaction more important since it is fundamental. The validity of many other checking axioms will rely on these axioms.

A. Logical implication

Intuitively, either something happens because of an interaction or because of it something does not happen. Similar things are true of its negation.

Formally, denoting an interaction between objects x and y as I(x,y), there are events E, E_1, E_2, E_3, E', E_1', E_2', E_3' such that

$$\neg I(x,y,t) \rightarrow [\neg E(x,t) \vee \neg E_1(y,t) \vee E_2(x,t) \vee E_3(y,t)] \vee$$
$$I(x,y,t) \rightarrow [\neg E'(x,t) \vee \neg E_1'(y,t) \vee E_2'(x,t) \vee E_3'(y,t)].$$

Our formula looks a bit clumsy, but that is necessary. The reader should also note that interaction can be composed by using logical connectives as long as each component interaction involves the same two (possibly composite) objects. The negation, conjunction and disjunctions of interactions are all interactions.

There are some subtleties to be mentioned. According to the axiom, in the case of continuous time, the presence of a certain interaction would make events happen infinitely frequent or in a discrete time coordinate system a different event would happen at each observing moment. But there is no serious difficulty in theory because those events can be viewed as subevents of a composed event with a long enough duration at a proper abstraction level.

Bidirectionality of interaction

An apparent problem, however, is that the asymmetry in the notion of causation is lost. The power we captured in interaction seems to work both ways. For example, if we say the locomotive pulling the caboose is an interaction then this interaction not only makes the caboose move faster but also makes the locomotive itself move slower. The reason for this is very straightforward: The interaction, as a relation, identifies the joint actual state (in some aspect) of both the objects. The state of chair pressing the floor is the same (it is more than logically equivalent) state as the floor supporting the chair. To say the locomotive is pulling the caboose is equivalent to saying the caboose is pulling the locomotive in the opposite direction. Similarly, if an apple is falling to the ground due to gravitation, the interaction between it and Earth, then the Earth is moving towards it for the same reason.

Our definition (and the ordinary sense) of causation has made us believe that there is such a thing as causational unidirectionality. Namely, if there is something that can be a cause, say I(x,y)– the interaction between x and y, then either x causes y to do something or y causes x to do something. One of them identifies with the cause and the other with the effect. The fact that interactions do not work this way makes some people doubt its status as a legitimate candidate for "cause." [Bunge] tries to use the notion of "causal approximation" to fix this problem. What he says is that although an apple and Earth attract each other, it is the movement of the apple that is noticeable to us and therefore, as an approximation, the gravitation does not lose unidirectionality and still qualifies as a cause.

We point out that the notion of interaction does not entail bidirectionality as far as causation itself is concerned. The problem here is to distinguish what are the arguments of a causal judgement. Our definition has made it clear that it is the state or event rather than the participating objects which are legal and sensible arguments. To make sure the causation is unidirectional, we only need to show that the one relation or event is caused, or induced, or forced, by another relation or event. If two objects each causes some changes to another, the both changes can be uidirectional. In the locomotive and caboose example, the change of movement of either object is caused by the pulling of another object. Therefore there is some abstraction process when we describe causal behaviors, but there need not be any approximation or neglect.

Unidirectionality in behavior description

Although we "explained away" bidirectionality in this case, we should give an explanation as to why in many other circumstances different objects are associated with cause and effect and why *the cause situation seems to only have effect on one object but not the other.* [14] Let us examine the example of lighting the match. Looking closer, we can see there are at least four subevents. For each of them, we can identify an interaction that is present for the duration of the subevent.

The first is "holding"

Hold (Hand, match),

i.e., someone's hand moves the match towards a surface by holding it. The interaction is between Hand and match.

The second is

Rub (match, surface).

Although the hand will be still in motion, there comes in another more important aspect. The head of the match gets overheated and ignites. The subevent is the ignition of the match. The interaction is the match rubbing against the surface. Supposedly the interaction changes location within this period. Or more accurately, we got interactions differing by their exact location for each moment.

React (mat ch,oxygen),

causes the subevent that the head of the match burns with the support of air. The interaction is that the chemicals on the match head stand in some state explainable in terms of ciffinity and its like. Again interactions change with each instant.

Filially, there are interactions between the shaft and the air, another reaction:

React'(flame, match shaft).

The event is that the flame on the head of the match burns the match shaft.

What this analysis suggests is that a sustained interaction (fixed in all other aspects except time) is rare when changing situations are involved. We usually have interactions for a sequence of instants, differing in location, intensity and so on. This analysis also shows that the interactions present in the four subevents are not suitable for conceptualizing even

[11] There arc linguistic reasons for this. Originally, the cause events are often agents' actions. But that is not our concern here.

something of such modest complexity. It is much more convenient to talk about the state when the match head gets hot enough to ignite, or the momentary state when the flame is strong enough to put the wooden part on fire. In short, talking in terms of events and their pre- or post-conditions.

For those concerned with the "causal power" of interactions, it should be clear that all the subevents are made possible by these interactions. Furthermore, they are not the only ones. These interactions also cause something to the hand, to the surface, and to the surrounding atmosphere, but that is not of interest for the person making the original statement. He has singled out a *"causal chain"* from a "causal network" due to the interactions.

This "causal chain" is made up of numerous interactions as we explained. The separation of that causal chain is possible and justifiable because we can identify some objects (which are of interest to us) that lie at the two ends respectively. And it is in this way that the second sense of unidirectionality (unidirectionality based on event causation) is derived from most causal statements. The first sense of unidirectionality (regarding associated objects), i.e., agency and patiency, in this view, depends completely on which changes (events) these objects are associated with, the preceding one or succeeding one.

We make an observation now. Interactions are mostly fine-grain sized causes of events. They do not directly play the role of cause in a complex causation.

Lastly we mention that the implication axiom also reveals another difficulty with interaction which is not as readily overcome. That is, it may allow apparently *simultaneous causation.*[15]

[15]In the axiom, we used the same time variable t (implicitly universally quantified) for both the events and the interaction itself. This is not necessarily an evidence for an apparently simultaneous causation. This only means that the interaction "immediately" precedes the start of the events. The interaction is distinct from the events because it is observed momentarily while the events have their duration.

The problem arises when the interaction is an extended one, i.e., its duration exceeds the small time quantum in our frame. We have argued earlier that actions made upon each other by interacting objects will not take effect "simultaneously". If we had a continuous or fine enough time frame, we could always point to the distance between the interaction and its delayed effect. However, suppose we have an event whose duration spans from time points 1 to 3. The interactions will then be observed at both points 1 and 2. (There is no ground for observing the interaction at point 3 since that would have nothing to do with the event which has already terminated there.) Although we know in general that simultaneously causation is impossible, notationally the only way we can express our observation would be to say that the interaction and the event start at the same time.

B. Physical contact

The power of interactions in causing something to happen or not to happen comes from the concrete properties of particular objects. The types of interactions differ from individual to individual. For example, a match struck against its box would be ignited but a piece of straw that is struck would not. Because they are so diversified, in identifying or locating an interaction, to rely on these particular properties is not always possible or plausible. There is a general property of, interactions that can be made use of for our purpose. That is its spatial property which is characterized in the following three axioms. Each of them is more specific in locating an interaction.

1. Adjacency

   This axiom asserts that interacting objects should be in close contact.

   I(^{x}>y) -^adjacent(Location(x),Location(y)).

   where the predicate adjacent is defined for point sets P and Q:

   adjacent(P, Q) =

   3pE P , qG Q [adjacentPoint(p, q)].

   The predicate adjacentPoint is defined as

   adjacentPoint(p,q) = [p=q V p€ neighbors(q)].

   Recalling our notion that space is discrete and the function "neighbors" of q returns the small cubes touching q, this part formalizes the notion of physical contact. This is based on the fact that every physical object has a location.[16]

2. Locality

   This axiom asserts that only those parts that are in contact contribute to the interaction.

   Let F/,Q/ be two point sets adjacent to each other, i.e.,

   VpEP/ 3q£Q/ [adjacent(p,q)] A

   VqEQ/ pGP/ 3p€P/ [adjacent(p,q)],

   and let *I[(xj,yj)* be the interaction between the parts of x and y occupying the point sots. We will have

[16] **We note here that the rejection of distant action only buys us theoretical uniformity but does not make the actual processing any easier. For example, if we deal with field forces we may not simply take the surface of objects as the physical contact they have. But this need not be as elusive as field force. When we blend two kinds of liquids, the contact points would also be hard to define.**

$I_I(x_I, y_I) \equiv$ I(x,y)

This axiom makes the interaction more specific. That is, it is on the actually contacted parts of two objects that there exists an interaction, and this is the *only possible interaction.*

3. Additivity

   It is important to note that in the axioms on locality and adjacency we have been taking all the interactions involving two objects as a whole. The axioms only holds for the interactions in their totality.

   The interactions between objects can be of any type even at the same contact place. For example, holding hands can have interactions that are both mechanical and thermal. The following axiom only specifies the additivity in terms of physical contacts.

   Let P'_I, P''_I be adjacent to Q'_I, Q''_I respectively.

   Also let $P'_I \cup P''_I = P_I, P'_I \cap P''_I = \emptyset$, i.e., P'_I and P''_I are two disjoint point sets whose union is P_I.

   Similarly let $Q'_I \cup Q''_I = Q_I$ and $Q'_I \cap Q''_I = \emptyset$.

   Let x', x", y', and y" occupy P'_I, P''_I and Q'_I, Q''_I respectively.

   Let I' and I" be interactions between x' and y' and between y' and y" respectively. We will have

   I'(x', y') $\wedge$ I"(x", y") $\equiv I_I$(x,y)

5.2.4 Non-physical Causal Relations

As we mentioned in the taxonomy of relations, in our model there is another category of causal relations: social relations. Typical examples are "possess", "own", "owe", "having right of way over" and so on. We have also mentioned that they have causal power and are more than their spatiotemporal correlates.

The nature of social relations is very complex. It is not possible to give a list of axioms that illustrates most of its important properties as we did for interaction. The few formulas we have are listed below. We note that an adequate treatment of agency is necessary for understanding this issue. This is given in the section on agents.

Assuming B(x,y) to be a primitive social relation (B for binding), there are the following axioms:

1. Agentiveness

   $B(x, y) \rightarrow \exists A[\text{agent}(A) \wedge A \in \arg(B)]$,

   i.e., $A = x \vee A = y$. This reflects the fact that all social relations involve agents (though, not necessarily persons).

   But this does not hold for composite relations. For example, occupiedBy(Table, A) meaning "Table" is occupied by agent A can form an existential abstraction

   $\exists x[\text{occupiedBy}(\text{Table}, x)] \equiv \text{occupied}(\text{Table})$.

2. Cause for agentive events

   $B(x,y) \rightarrow [\exists E[B \Rightarrow E \wedge \exists A[\text{agent}(A) \wedge A \in \arg(E)]]$

   This states that B is the cause of another event and identifies the effect to be the action of an agent. This requirement is the result of our operational approach, i.e., any event should have observable consequences.

   This causation, as the nature of the effect event suggests, has an unobservable intermediate situation in its causal chain, namely, a mental state. Without this part, the property of continued causation would be violated. On the other hand, in our operational theory, no assumption of particular mental states is allowed. We fix the theoretical difficulty by relaxing this restriction a little. Mental states can exist if they are associated with social relations. They are assumed to exist as soon as the agent involved has recognized the relation.

   We note that these kinds of mental states are a small, much restricted subset of all possible mental states. In the sense that it leads to a fixed and uniform reaction, a mental state invoked by a social relation is fixed. For example, knowing that one owes money to someone, one will usually pay back. On the other hand, mental activity in general, and mental states invoked by abstract relations in particular, do not lead to uniform reactions for all people. So this is also a criterion in distinguishing an abstract relation from a social relation.

   Admitting mental states does not suggest that a social relation is mentalistic and subjective in nature. As a matter of fact, any such relation is independent of the knowledge of the agents involved in it, although an agent can act only when he has the knowledge to.

3. Physical basis

   $B(x,y) \rightarrow [\exists I,R[B \equiv [I \wedge R] \wedge \text{interaction}(I)]]$, where R is a non-causal relation.

The equivalence means that whenever such physical conditions hold, the social relation also holds. But ontologically the social relation is completely different from the associated physical conditions. For example, whenever my signature appears on a check, I am responsible for paying the amount specified on it. On the other hand the sheer appearance of a check is different in nature from my financial behavior or state.

This axiom is fundamental to the notion of social relation because it provides a way to indirectly observe it.

However, because R can be any non-causal relation the observation may be complex. For example, for the social relation motherOf, assuming we are talking about biological mother, the formula is

motherOf(A,B)≡OgiveBirth(A,B),

which might require us to go back deep into the past. On the other hand, interactions are by definition instantaneously established.

4. Social basis

The social basis of a social relation states that assuming in a particular domain Dom there exist an event E, non-causal relation R, and interaction I such that

B=>E and B≡[I A R],

it is possible to have another domain Dom' where all natural laws hold and B=>E but no longer B≡[I A R].

If we can assume all interactions are known to us then the clause "all natural laws hold" may be replaced by a more "mathematical" formulation. E.g.,

[Pi-QiA Pi.QiS W]->

[3P_2, Q_2[P_2, Q_2e W' A P!=P_2A Q^QsA P_2->Q_2]]

It is very hard to directly establish the essence of a social relation as we did for interaction. The above axiom (not in formula form) is constructed by playing a trick with possible world mechanism (a domain is like an instance of possible worlds) in order to give a "non-constructive criterion" of a social relation. What it essentially suggests is that the way social bindings are set up is somewhat arbitrary. To the same physical situation different social meanings can be attached. In different possible worlds the same social relation may be entailed by different physical situations. On the other hand, in different possible worlds there can exist the same social relations which means that they are causally bound to the same events. For example, both in the British islands and the continent there is the so called "right of way" traffic rule, although the physical conditions may differ.

We now examine if this characterization is consistent with our general notion of causation.

1. Arbitrariness of mental states

   One doubt concerning the legitimacy of social relations being causal is that they are under the influence of mental states and mental states can be arbitrary. For example, not everybody owing money would pay back: one can be delinquent or file for bankrupcy.

   This argument is not convincing. We observe that people in debt regularly pay back. If this regularity has exceptions then all natural phenomena have exceptions, too. A free body may not fall if it is in weightless space-- The examples abound. All we need is a "*ceteris paribus*" clause, which is the case for causal statements in general, or an abstractional model for the case of formal specifications.

2. Locality of social causation.

   Like temporal continuity, if no special provision about mental state is made, spatial continuity of causation would be in jeopardy, too.

   There would be no problem if the causal link could be viewed as only concerning the individuals, e.g., if the owing of money by someone were solely related to him and his debtor. Among the factors relating to a social relation the recognition of the relation is local. The acts of the agent based on the knowledge of the relation would be physical and localized, also. However, as we noted, a social relation is supposed to derive its power from something external to the individuals. As we mentioned in the section on taxonomy of relations, it is a phenomenon originated from the collective behavior of a set of individuals. It is because of this collectivity that social relations are uniformly recognized and responded. We have suggested that this external collective may establish a social relation through speech acts or other communication and cooperation measures. But how this is done is a big Unknown. It is hard to speculate on the way this Unknown affects how the agent is localized.

5.2.5 The Role of Causation in the Operational Model

The notion of causation plays a crucial role at two levels: as the theoretical foundation of our descriptive calculus, and as the basis for various specification validation techniques. [17]

1. **A foundation for our descriptive framework.**

   First, our intuitive feeling for the completeness of the description of a system is rationalized by the concept of causal connection. Without the notion of causation the concept of operational completeness (cf. chapter 2) would not find a ground.

   The objects of our description are not always causally related. Some are simply *abstract objects and beyond any causal force*. Some are just *correlates of the same unknown or complex cause*. Some are *part of the goal* structure (cf. chapter 7) stemming from minor human needs. Some changes may be treated as self-movements because we do not want to get into the detail by going inside the objects. As far as system operations are concerned these changes are not caused and they do not cause anything. Even for those causally related, the complete cause may not always be expressible.

   However, as to our second point, the majority of observable behaviors can be put into causal perspective. In particular, we mention the two descriptive primitives in our scheme: precondition and consequence.

   According to our definition of causation, P is a precondition of E if and only if P is an INUS condition of E or part of the cause for E. Formally,

   $\exists R\ [P \land R] \Rightarrow E.$

   This can be easily verified by our intuition. For example, we know that

[17] Causation plays an important role in building our theory. In a sense it is the center of an applied ontology.

For any ontological enquiry, the notion of objects and relations would be the first to be examined. And this is rightly so since they are what the world is. But discussions purely concerned with questions like "what are objects or events?" etc. stay at a level where logic and technical analysis can not get in. This is because we can not easily get those various concepts to interact with one another.

Causation is a subject that brings in interactions among concepts, both in terms of their conceptualization, and their logic forms. For example, the difference between abstract (e.g., mathematical) objects and physical objects is best illuminated in light of causal significance. The investigation of the nature of "unchange" (negation of events) makes our understanding of events more complete. In general, if in broad ontological thinking there is no place for "event" for many authors, then on this subject, the term "event" is the common key word since this is what causation relates. In terms of technical details, the logic forms of events can be evaluated in terms of how they serve the formulation of causation. Taking events as predicate formulas proves advantageous because we can then easily formalize the logical aspects of causation ([Mackie] and [Burks]) and all the causal statements can now be processed in systems based on first order logic.

The notion of causation deepens our understanding in general, too. For example, the difference between a logical (material) consequence and an ontological one is made most clear through the discussion of causation.

$$\mathbf{P} \prec_{\mathrm{I}} \mathbf{E},$$

and

$$\neg\mathbf{P} \rightarrow \neg\mathbf{E}.$$

Similarly, the consequence Q of event E is operational rather than just logical. Formally,

$$\mathbf{E} \Rightarrow \mathbf{Q}.$$

Similarly this reconciles with our intuition about the notion of consequence or post-condition fairly well.

These immediate causations, when connected together, form *causal chains* which are the foundation for the very idea of simulation. We point out that the actual simulation process, in inferring preconditions, has to make use of general constraints in the form of axioms. These constraints are essentially correlates. But the main content of a simulation is following causal chains.

2. **The** basis **for the** analysis **technique.**

Our analysis of specifications is mainly a causal one. Therefore, each rule in the analysis is based on the notion of causation. Here we just give an overview, emphasizing the link between the causation principle and the concrete checking rules. We go through several groups of examples.

(1). The genetic principle of causation found its way directly in the checking rule "nothing happens to non-existing objects" and other similar rules. Another straightforward use of causation is the locality principle of causation. A heuristic rule in checking can be: Make sure the spatial locations for objects involved in the same physical event are in contact.

(2). The precondition and consequence as defined above have somewhat unexpected implications. Namely, since precondition is the immediate cause (or part of the cause) it should be "connected" with the event as the predicate is defined in this chapter. The formal consequence is that, assuming all events and relations are primitive,

p(x,y) is not the precondition of event e(u,v)

or

p(u,v) is not the consequence of event e(x,y),

if x,y and u,v are distinct.

Apparently, those can servo as good chocking heuristics.

(3). There might be cases where one finds it more convenient to specify correlates than the direct cause or effect. And we may relax our requirement on the actual specification of the precondition and the consequence a little bit. For example, the consequence of a customer ordering some dishes may be specified as a social property of the customer (contingent upon his being in a restaurant), e.g., "outstanding order" or "current order". It has the value "order" right after the event. This is clearer than saying that some writing is on the waiter's notebook or, even worse, there is a change in the waiter's mental state.

But we note that the converse is more important. Namely, to avoid mistakes we should put in the direct causal relations rather than their logically equivalent correlates. In fact, most people would put the directly involved objects of events in the argument lists for their specifications simply because this is more intuitive appealing.

Theoretically, causation is the basis for teleological explanation. In our checking, the teleological constraints are based on causal reasoning. For example, teleology tells us to eliminate event paths that do not contribute to the achieving of goals. The unnecessary paths or events are picked out by examining individual causal chains. If a path or event is not on a chain with specified goals it could be eliminated.

(4). The above mentioned constraints are themselves causal constraints in nature. We also have constraints regarding other aspects of systems, in particular, the stability of a system. Here, causation principle is the basis. For example, a property of stable systems is that they restore their original states for each recurrence of their behaviors. In the actual checking, to detect if a certain state is restored and if the restoring path does not violate other constraints, these original states are treated as a goal state which is then checked as in the usual teleological constraint checking. Another examplar property is the going up and down of system resources. The notion of causation is at work here, too, since the foundation of this property is that the genetic principle can not be violated.

5.2.6 Causality and Teleology : Behavior vs. Function

Teleology : semantic **or ontological?**

One may argue for the semantic nature of teleology because we know there are other events and states that are causally related to goals but are not mentioned in the description. That is to say, it is only the decision of a particular observer to group a scries of phenomena together that makes a teleological account of a system. On the other hand, one may point to the fact that the survival of a living organism is *actually* dependent on those events identified by a teleological explanation. Therefore teleogical accounts make "real" sense.

This is a long standing problem and we do not expect it to be settled here. What we are sure of is that the notion of teleology is based on the notion of causality as far as behaviors at the same abstraction (description) level are concerned. Once one starts talking about higher level goals that are stated in terms beyond the original level, semantic problems would come in.

Motivation for Searching for "Pure Behavior"

Again, from our engineering needs, what we would like to do is to find out how teleological considerations may affect our specification activity.

The intended use of a system or its components and the environment where they are intended to be used often form hidden assumptions. A description generated without realizing these assumptions, i.e., hidden constraints, tends to predict wrong behaviors when the assumed conditions break down. For example, one may specify the consequence of the flipping the light switch to be that the light is on. However, if there is no power supply, even if the switch is flipped, the light will not be on. The problem also exists when specifying social processes. For example, one may specify the consequence of an agent's selling something as "possessing" the payment for the merchandise. However, unless the payment is in cash, he or she only possesses a piece of document.

In describing systems we would like our descriptions to be free of hidden assumptions and therefore modular.[18] A modular description allows us to use the same description for an object or system at different places. [19]

But a description is always written in a certain context. An action by an agent always has certain goals which are intended consequences. If we are asking for pure behavior then we have to consider two theoretical questions : (1) Whether there is such a thing as pure behavior, (2) If the "purity" of a behavior is relative, when can we know that we are close to the limit, i.e., what is the criterion of "pure" behavior? The answers to these questions will help us to write understandable and modular descriptions. This is the motivation for this section.

[18]On the other hand, the intended use or goal of a system or its components is often the organizing theme for a description. This positive role of goals will be discussed in the chapter on systems.

[19]The principle of No-Function-In-Structure [De Kleer 1984b] is a specific way of achieving modularity.

Structure, Behavior and Function

Parallel to the dichotomy of causality and teleology, we have the dichotomy between behavior and function in system descriptions.

Many AI researchers make distinctions among structural, behavioral and functional descriptions. In our perspective, "structural description" is really the description of the behavior constraints on the individual parts and interactions among these parts. "Behavioral description" is the behavior of the whole which is equivalent to the total behaviors of all the parts.

The key difference is between behavior and function. It is not obvious that there is a clear demarcation. We maintain there could be one, but somewhat relativized.

By pure "behaviorial", we mean that a description only contains objects that are directly observed during the event, or, more accurately, observed in *direct contacts*. In the terminology of our causation, it is based on "immediate causation". On the other hand, a functional description is more than "direct observation". It involves interpretation in various ways. In particular, it allows one to assert an event as a result of a causal chain. Or it allows one to "group" a sequence of events together and talk about the new event at a higher abstraction level.

For example, if someone is said to have turned on a light, the description is functional. This is because in this event description, the two arguments, the person and the light bulb, are not in direct contact and therefore a *necessary condition* is not met. On the other hand, the event of the person's moving a convex part of the switch is purely behavioral because they are in direct contact. Due to this direct contact, the happening of the moving of the person's finger is logically equivalent to the moving of that part (Although the former is the cause for the latter.). On the other hand, even if that part is moved, the lighting of the bulb is contingent upon many other conditions.

Pure Behavior and its Relativity

Questions may arise for a statement such as "agent A turns on the switch". There can be two interpretations for turning on the switch. If the interpretation is that A moved a part on the switch as explained above, then it is behavioral. If it is to mean that a connection is made inside the switch then there are again two cases. If the switch is understood as made up of several parts then the making of the connection is contingent upon the normal working of the internal mechanism. Therefore this should be a functional description. However if the switch is treated as a whole, it is impossible to have any internal breakdown. Then this again is a behavioral description.

As a most general statement, it is always feasible to equate contacting a part of a rigid or unbroken object as contacting the whole. We may take this as the definition for the "absolute" purely behavioral description. We note that this is fairly consistent with our intuition of "pure behavior".

To a less general degree, if several objects are spatially contiguous, their behaviors are causally related, and that it is assumed that there is no internal breakdown, one can take them as a whole. Contact to any part would be contacting the whole. A description of such contact is behavioral. Of course, there is a limit to what we can believe as never broken in certain circumstances. For example, in the case of turning on the light bulb, both the spatial connection and the causal link make it unlikely for us to believe that they can be treated as pure behavior. We may call this contingently behavioral. This is often what we would like to do in practice.

In any event, there is no infinite regression in terms of the causal chaining, as the regression will stop when we have immediate causation.

However, there is another notion of infinite regression, which is in terms of the grain-size of our observations. One may argue that there is no really such a thing as finger or plastic knob, – what we have are atoms, molecules or quarks.

Here, one is making statements about exactly the same behavior, only at different levels. No doubt there does not exist an a priori "right" level for the description of any event. In this sense, the purity of a "pure behavioral" description is relativized.

A functional description can be both a semantic abstraction, i.e., a statement at a higher level, and a causal abstraction, i.e., the final joint on a causal chain. For example, in terms of world history, we are justified to say "Hitler invoked World War II".

In the current version of our operational analysis, we are solely concerned with behaviors described at the same levels, i.e., the purely ontological aspect of teleology.

We note that we then do not need to introduce any new formalism, as the handling of causations has it all.

The intuition of teleology is that all the behaviors of a system should be explained by a set of events and states designated as goals. Other events are to achieve these goals. A formal treatment of this intuition will be presented in a later chapter on system models.

5.3 The Notion of Causation in AI Research

The interest on causation in AI has been mainly associated with the prediction and explanation of the behavior of physical systems. [Rieger and Grinberg], [De Kleer and Brown], [Forbus] and [Kuipers] all used some causal notion in their work. Among the works published, research done by De Kleer et al. has been profound in conceptual depth, technically sound, and influential in the field. We will take their notion as representative and conduct an analysis and comparison.

5.3.1 An Analysis of De Kleer's Notion of Causality

Our analysis will be based on two articles presented in a special volume of AI journal. There are technical works, theoretical analysis, and broader methodological claims. We will concentrate on the technical and theoretical analysis and only briefly comment on the more general claims.

In the work of De Kleer et al., the notion of causality is expressed in the form of the properties of certain hypothetical information processors. In their words, each of these processors,

"(a) has limited ability to process and store information,

(b) can only communicate with processors of neighboring components,

(c) acts on its neighbors which in turn act on their neighbors, and

(d) contributes only once to any particular behavior for each disturbing influence."

But according to the authors, this corresponds only to a "classical " notion of causality. This notion can only explain the interstate behavior of systems. To be able to explain intrastate behavior, we have to "extend the architecture" of the processors. Consequently a notion of mythical time is introduced, during which a system transfers from one equilibrium to another and no real time passes. During that mythical time, a "causal process" takes place. In this process, the processors, being able "to distinguish between a new equilibrium value from an old one", set new equilibrium values, one by one, until all the values are right for the new equilibrium. The computation is based on the confluence equations derived from quantitative descriptions of systems. When there is more than one unknown in the equations, a processor makes a guess using "canonicality heuristics" so that at each computation step, one value is changed and *fixed*. This is thought advantageous over solving simultaneous equations because it follows the same basic causal principles as at the interstate level and results in a pure causal explanation or prediction of system behaviors.

Our analysis will be carried out in three steps: on their "classical " causality, on mythical time, and the "causal process". To do the analysis, we first "translate" the information processing language to our ordinary ontological terms. This is fairly straightforward. As **a** matter of fact, most of our "translations" are the alternative versions used in the papers.

A. The "classical notion" of causation

In a non-information processing formulation, the notion of causation is that: Components' behavior changes only when being acted upon. The actions take place locally, i.e., only neighboring components can act on a component. The action is unidirectional, i.e., the action of one component will bring a disturbance to another component but not vice versa. To put it another way, there is only one interaction (in their sense-K.Y.) per disturbance. As De Kleer summarized, there are basically two principles in this: locality and unidirectionality.

First we notice that the causal arguments are only events (changes or disturbances). Therefore this makes absence of change or state (e.g., preventing something to happen) not a cause or effect. But this is not a serious problem. In the problem domain they are concerned with, it seems all right. On the other hand, this makes many complexities go away. For example, we can now have strict precedence between cause and effect, which is exactly what they have in their theory.

Secondly, from our perspective, because they assert that components change only when being acted upon, their notion has almost explicitly stated the principle of no self-movement. The locality principle is also clear and emphasized. That cause and effect are to be associated with distinct objects is also implicitly assumed since they have actually a much stronger requirement. As for the logical implication of causation, it is clearly a "sufficient condition". It is built into the processors' computation process. It is also implicitly assumed that the causality they talked about is causal laws.

There is one single important difference, however. They have equated the asymmetry of cause and effect with the unidirectionality of actions between two objects. This unidirectionality is equivalent to saying that the action of one component will not have a coexisting reaction from another component. As we explained in the previous subsection, this does not hold.

Of course, we do not care whether Newton's second law is literally followed in a description. We do say that the cause of the moving of the carriage is the pulling of the horse. Similarly we do not usually mention the Earth's perturbation when an apple is falling. *And it is exactly this kind of naive observation that cultivated the kind of causation notion in naive physics,* which people are accustomed to and quite comfortable with.

This notion, however, is not appropriate for any mutual action involving objects in approximately equal conditions, e.g., in the collision of two rigid bodies or for so called many body problems which is an area of intensive scientific research.

B. Mythical time

On this issue, the first remark one may make is that even processors need time to compute the new values. The assumption that information processing activities do not take time is not well grounded. Apparently the authors are aware of this extreme difficulty and that's why they use the term "mythical". But this name does not make the difficulty go away.

In a real system the number of states is larger than the number of equilibria. There are temporary states for transitions from one equilibrium to another. The phenomena involved in this kinds of processes are fairly thoroughly researched.

If you assume discrete state you must assume a discrete time scale since, just as the citation in the same section says, "Time and space are not things but order of things" —Leibniz. In most discrete time schemes you just assume that the states are undefined during a transition. (It is a different question how the state is changed. We now just care about when.) It is important to point out that in many circumstances this approach does not leave us many undefined states. If the interactions that force state changes take relatively shorter time, for example, the hitting of a person by a car and his injury, then the transition period would be neglegible. Most of the time the objects are in certain defined states. We observe here that exactly because of this phenomenon people's naive notion of causation assumes no time gap in state transitions.

Physics and calculus take another route. They make both continuous and thus the causality notion also works out well.

If there is a need for "mythical time" then it is an internal problem with their notion of causality. It tries to deal with continuous process with discrete qualitative states using continuous time. The need to create an illusion that every moment is accounted for makes it mandatory that they do not allow time for transition from state to state. Again this is the need for an explanation in naive physics but not an inherent problem with causality.

C. "Causal process"

The implication of "causal process"

The translation is about as straightforward as before: The causation here should take the same form. That is, one change causes the other, locally and unidirectionally. In particular,

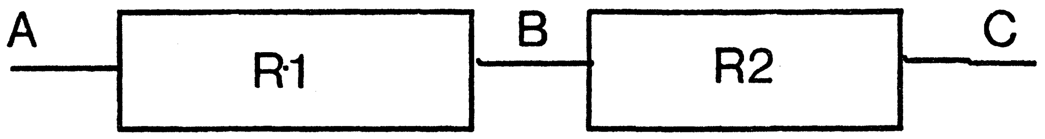


Figure 5.1: The Resistors

the changes are exactly that the state of the component transfers from the old to the new and this transfer takes mythical time, i.e., no time.

Take a simple example. Suppose we have a resistor connected to a power supply as in figure 5.1. Suppose the voltage is changed from V_0 to V and the current from I_0 to I. Also suppose we know the change of voltage is the cause of change of current. (It is the opposite in the case of battery. In the case of resistors, exposure to light or heat can also has effect in the opposite direction.) What this suggests is that the voltage will change to V and during all this time of voltage change the current remains I_0. The moment the voltage reaches V the current immediately reaches I.

If one is not quite familiar with electricity, imagine a pump. The fluid would not start moving until the hand of pressure gauge stops moving. This is essentially the causal picture that is depicted by the principle newly introduced in "causal process".

From the pump example one may realize what should be going on as physical reality: The actual state transition is achieved through the mutual actions among the pump, the fluid and the conduit. Their momentary states change simultaneously as a result of mutual momentary action and reaction and through a certain period of time.

Of course, the authors are not ignorant of this consequence of their theory. They point out themselves that "*it is the violation of the confluences that results in causal action*", where confluences are some mathematical expressions of physical laws concerning equilibria. In other places, it is similarly commented that during this mythical time, the laws governing equilibrium conditions are violated.

Our point is that to assess the severity of their claim, we have to be aware that not only the laws for equilibrium are violated, *all laws of physics have been violated.* (For any instance we can verify that there can not be a moment that fits the above suggested state configuration or state value assignment to an equilibrium equation or equations describing transitional states.) Therefore, there is no physical reality whatsoever behind that causal notion. On the other hand, the same phenomenon could have been given a complete causal account, according to known physical laws.

The applicability of this notion

Next we would like to show that the applicability of this notion is very limited. Not in terms of circuit analysis, perhaps, but in terms of serving as a viable causation principle for a moderately general domain.

First, we note that the possibility of utilizing this causal process along with the heuristics is based on the relative speediness of the state transition with respect to the lasting equilibrium of interest to us. If the resistor is big enough, e.g., our Earth, not only the "component model" would have to change, but also this causal process would not work.

Secondly, the reason that we can have those nice heuristics is partially because the systems are designed with purpose and modularity, i.e., good engineering. The circuits and spatial layouts are arranged so that two equally stable states can not be achieved (under influence of thermal or environmental noise perhaps) from the same old stable states. Otherwise they are just bad designs and not workable. Most would be forgotten before we get to see them.

Lastly, the prerequisite for a sensible account in this line is that we are only concerned with *macroscopically observed equilibrium*, formed by *continuous medium*, be that hydraulics, electronics or thermodynamics. When the explanation has to be microscopic, e.g., when explaining the behavior of travelling waves or when the old equilibrium is not made to transfer to a new equilibrium but to a further chaos (e.g., a chain reaction in an atomic explosion), it is the process but not the end result which has to be accounted for. In that case, the causality designed for equilibrium is completely inadequate.[20]

[20]There seems to be a profound reason why most systems familiar to us stabalize quickly instead of exploding. My speculation is that they all have an elastic or quasi-elastic medium which "implements"

Alternative

The task set forth by De Kleer et al. is to generate a causal account at a level where it can in no way be "causally generated". Undoubtedly the difficulties revealed in our analysis are the prices they decided to pay for this task. The problem is then how worthwhile is it to produce a "pure" causal explanation.

As many philosophers ([Russell][Bunge]) point out, causality is not the only lawful relation nor the only basis for scientific explanation. Functional interdependence is more general. And there are other lawful relations, such as statistical laws, as well. Actually, as demonstrated by De Kleer et al., a behavior prediction need not be causal to be convincing.

For ordinary researchers in physics or engineering, it is also true that *reduction ad absurdium* is often preferred "psychologically". In fact, there is a personal preference with respect to the use of network theory versus equivalent circuit models when circuits get complex. Some people find the latter more rational and tractable.

We would suggest that an alternative could be taken. Essentially the same state transfer heuristics can be given new grounds and new interpretations. The interpretation is lower level mutual actions among media. The grounds are design principles and special system characteristics. The state variables now simultaneously achieve new values through bidirectional causation.

5.3.2 Comparison of Roles and Contents of the Two Notions of Causation

The project De Kleer et al. work on is reasoning *from structure to behavior*. What De Kleer et al. take as given is a structural description. What they would produce is the prediction of the behavior and function of the system. That is, they predict behavior and *extract* a causal account of it. They work with behavior and generate causal explanations or descriptions.

We are also engaged in producing system descriptions but the descriptions are manually written by users. Our influence on the descriptions is the framework we impose through the language. We then check a causal description, or causation-oriented description. The information on causation is directives in checking the correctness of a system specification.

Both systems intend to produce understandable descriptions of system behaviors. Both find causation an important cognitive dimension to exploit in this regard. Both face the difficulty that the naive notion of causation can not give a satisfactory explanation for all of our observations. In their case, the typical case is the transition from one equilibrium to

mutual actions better. However this is pure speculation. In any events, physics is able to account for both stable and transitional behaviors without the mythical causality if it can explain at all.

another. In our case, a typical example is the social effect of one's physical action where a network of mutual actions exerts a certain "force" on the relevant people. Both theories seek a solution that accomodates or preserves the notion of causality. For them, it is the theory of "causal process". For us, we introduced the notion of "social relation" without having a set of axioms to identify or locate it[21]. But the commonality ends here.

Theoretically, they built a new theory to support their notion of causal process so that causation can be the basis of all explanations. The notion of "mythical time", say, is part of the theory. For us, we accept the fact that causal explanations are not the only form of scientific explanations. For the kind of causation we introduced for explanatory purpose, we admit their intermedite status and do not intend to give them any profound meaning in terms of causation. We believe they are just the synergy of a huge number of interactions engaged in by agents (in the case of social relation) or by particles (in the case of hydraulics).

The difference in the goals of two systems entails all differences in the *concrete ways causation is dealt with.*

For them, the technical emphasis is the algorithm and heuristics to *extract* a causal account. This is because the better the algorithm and heuristics work, the more likely that an acceptable explanation can be generated. The difficulty is synthesizing or generating. The competence of the system is judged by coherence of an account.

For us, the emphasis is on sharp definition or criterion for causation. This is because we need them as the ground for correctness checking. The difficulty is analyzing and differentiating. The competence of our system is judged by the soundness of the checking results on pieces of specifications.

The *domains* differ. As an engineering application oriented research, their domain is narrow (physical in a narrow sense, not including biological system, for example) but concrete. As a result, their conclusions can be easily tested. The domain that is treated is physical processes, which have been investigated as continuous processes thouroughly. As a result, their problem solving system processes both quantitative and qualitative information. As a general language design project, our domain covers both physical and social systems but only at qualitative level.

Because of the difference in functions, the contents of the two notions of causation differ. Several aspects are worth noting:

(1). As noted earlier, they only treat events as causal arguments. In fact, this is further reduced to variables in component confluences. Also noted is the strict temporal priority of cause over effect. But these are not substantial differences.

[21] **We succeeded in providing a, set of axioms for interactions.**

(2) The single profound difference is in how to view the interactions between objects. The asymmetry of causation is perceived by them as unidirectionality between "cause object" and "effect object". As we have shown, this leads to various conflicting treatments of concrete problems, including so called "mythical causality".

(3). The notion of causation in our theory is presented in a more general form. For example, we allow causal statements to relate two spatially non-contiguous events. (Only *immediate causation* requires spatial contiguity.) For another example, we included a genetic principle in our notion, which provides some powerful rules for our checking.

(4). Most importantly, we try to analyze social causation. In particular, we introduced the concept of information object to capture some of the causally significant social relations in communication and cooperation among agents. We also developed a simple agent model to make social relation a special case of causal relation.

Chapter 6

Formalizations of Basic Domain Categories

6.1 Objects

Formally, an object w is a physical object if and only if

$\exists$l:location l=LOCATION(w);

and

EXTENSION(w) undefined.

An object ψ is a concept if and only if

LOCATION(ψ) undefined ;

and

$\exists$S:set [$\forall$w$\in$ S object(w) $\wedge$ S=EXTENSION(ψ)].

The function LOCATION should return a value that is able to uniquely identify where the object is. The value of the function EXTENSION is, intuitively, the set of objects that match the intension of ψ.

We note that both functions LOCATION and EXTENSION may not be present in the abstractional model. Even if a particular kind of spatial relation is of concern, the exact location may not be relevant for an account of systems at the abstractional model level. Similarly, the value of EXTENSION may not exist at the higher level, if the specifier is only concerned with the abstraction, e.g., the content of a poem. On the other hand, an abstract object may arise out of the abstraction procedure itself and therefore it would not even exist at the lower level.

The realization of an abstract object is defined as its EXTENSION. A realization can be very general. E.g., the realization of a dish name stated without any context may be all the dishes that ever existed. A realization can also be very specific, e.g., the realization of "the person with name George Washington, who was the first president of U.S." is a single person.

A physical object ϕ is said to be an information object if and only if

$\exists\psi\exists$A:agent[ψ=CONTENT(ϕ, A)],

where CONTENT is yet another primitive, a domain dependent function. Intuitively, it is the concept that is denoted by the information object with respect to the agent.

We assume the content of an information object is the same to all agents recognizing it. Therefore, we can write

$\exists\psi$[ψ=CONTENT(ϕ)].

We note that CONTENT is a functional mapping. One may see a file as blocks of memory bytes, as a sequence of characters, as a piece of text, and so on. But "CONTENT" does not mean any of these things. It is reserved for what we would call meaning content. These things are abstract objects though, and the category for them is "symbol."

An abstract object θ_ψ is a symbol if for some mapping f:

$\exists\psi\exists$S $\exists\phi\forall\theta_\phi\in$ S

[physicalInformationObject(θ_ϕ) $\wedge$ CONTENT(θ_ϕ)=$\psi\wedge$ f(θ_ϕ)=θ_ψ],

In this case we extend the definition of CONTENT to say

$\exists\psi$[CONTENT(θ_ψ)= ψ].

Note that here we often do not have functional mapping because the physical form of an information object can be viewed as different symbols at different levels.

The realization of an symbol is the same as the realization of its corresponding concept:

realization(θ_ψ)=realization(CONTENT(θ_ψ))

6.2 Agents

The category of agent is of extreme importance for the description of real world systems. This is especially true with social systems since most events in a social system are actions performed by agents.

People have strong intuitions about agents just as they do about causation. While the image of agents is predominantly associated with persons, there is a consensus at a more general level as to what constitutes an agent.

An agent is usually thought to be an *active* component in a system. It can *initiate events*, i.e., cause something to happen to others without being directly acted upon itself. From this general observation, there are assumptions that an agent can have *goals*, can *form plans* for achieving their goals and have the proper *control* to carry out the plans. An agent is also able to *sense* its surroundings beyond the direct interactions exerted upon it by its surroundings. In a situation where there are many agents, they can communicate. In particular, they may communicate with symbols. That is, they can recognize the abstract form of an abstract object through a physical form.

If the set of agents is equated to the set of people, there is no need for a definition. Otherwise, we need criteria for distinguishing agents from non-agents. For example, we may want to have robots, computers, etc. on our agent list while excluding rocks, chairs or tables. As for vending machines, thermostats, and, perhaps, airplanes, different people may favor one or the other. Given that people have many seemingly very discriminating intuitions, however, a rigorous definition of agent or agency is very hard to come by.

In particular, any operational definition will fail. This is because external behavior alone is not sufficient for making the distinction. Take the initiation of events as an example. It is true that people spontaneously do what they do and rocks will not move unless being pushed. Namely, we observe self-movement from people but not from non-agents. However, there are cases where inanimate things change themselves. A rock may crack. Radioactive matter decays. Therefore the initiation of events is a criterion for some systems but not a valid one for all cases. Nonetheless, in practice, we use the initiation of events as a heuristic. And this "initiation" is operationalized by noting that the happening of one event does not immediately follow the happening of another.

In general, i.e., even taking internal structure into consideration, a clear delineation of the concept of agents is still a formidable task. Supposedly, an agent is to have some kind of control structure so that it can form and keep a plan, can command the acting unit to interact with surroundings. But it is hard to form a consensus on what constitutes "true" control.

We may assume that a composed object is exercising control if it does information processing, i.e., performs a complex transformation from input to output. The transformation can be as simple as automata, e.g., Mealy or Moore machines [Hopcroft and Ullman], or as complex as Turing machines. But for some objects, the control may be realized through mechanical or electrical interactions and there is no separate phase of information processing (as the term is used in its usual sense) or separate structure for it. On the other hand, we do not yet know what kind of information processing is going on in a human mind. Nevertheless, looking for a control component seems to be an attractive heuristic.

For our operational model, we will give only the following axioms that partially operationalize the behavior characteristics of an agent. They mainly reflect the communications ability of agents. In the following we will use Agent in two ways: predicate and function. If x is an object, Agent(x) means that x belongs to the class of agents. If the argument is an event e, then Agent(e)=A means A is the agent that performs the action e.

1. Only an agent can sense (i.e., acquire information without engaging in interaction with) other objects. If the class of sensing events is Perceive, then we have for all objects x

   -i3e [Perceive(e) A VxEParticipants(e)[-iagent(x)]].

   Formally, Perceive can be characterized as

   Ve[Perceive(e)->

   [[3A GParticipants(e) agent(A)] A [->3r[relation(r)A e=>r]].

   Of course, an event in an operational model will eventually cause something. That is in the second axiom.

2. Because of the purposefulness of an agent's action, the sensing of an object will eventually be reflected in the agent's later actions, i.e.,

   [Perceive(e) A A=Agent(e)A xGParticipants(e)]—>

   3e',y,r[eVe A A=Agent(e')A

   yGParticipants(e')A e'=>r(x,y)].

   Note that since r is a simple relation this only requires that an event happen to something that relates to x. The event does not necessarily happen to x itself.

3. As a particular case of general perception, an agent can recognize symbols it communicates with. This is reflected in the axiom on information objects given earlier.

4. Since, by definition, only agents can recognize symbols, only agents can manipulate them. This means

   $\forall\theta_\psi[e(\theta_\psi)\rightarrow\exists A[A=Agent(e)]$.

   Note that this e is an abstraction of a physical event. The axiom means that any event involving a symbol should be an agentive event.

5. As a special case of point 2, for a symbol,

   (Perceive(e)A e(A,0^)A Agent(A)]->

   3e\y,r,f [eVe A A=Agent(e')A

   yeParticipants(e')A e'$\Rightarrow r(f(\theta_\psi),y)]$.

Here r(f(θ_ψ),y) should be more restricted, as explained below.

We know that in most actual systems, r and f are interesting or meaningful relations. For example, f may be a function for realization and r an interaction. Secondly, just by the definition of information objects, we cannot allow the relation r and f to be arbitrarily general, because that would mean that some transformations at the symbol level might be done whereas we always assume that agents act according to the *content* (i.e., abstract content not abstract form). For example, although an agent may spontaneously decide what to do on the basis of whether a piece of text is written in English or French (regardless of what the content is), in our framework we restrict that his actions be linked only to what the content of the text means.

Even in the case of the content, in an operational system model, we would like to eliminate the arbitrariness of the decision an agent might make. For example, it is not desirable for an agent to produce three pieces of blank paper or three rocks when the command (θ_ψ) calls for producing three books. In this case, the transformation, taking the numerical value of one aspects of the content of the information object, is overly simple.

Due to the difficulty in handling relations in general, in our scheme we allow only r to be a causal relation (i.e., an interaction or a social relation). As for function f, it should not involve transformations at the symbol level but is otherwise not specified. The exact formulation will not be stated here.

There are other properties of agents. For example, agents are usually mobile. These properties can be used as heuristics for recognizing agents. If the agency is known, they can help in reasoning about the states of systems, e.g., the spatial relations involving agents.

6.3 Physical Events and Event Abstractions

Having given a rigorous description of how the objects stand in mutually influencing relations in the chapter on causation, we can now gain a clear understanding of what an event is. In this section, we will first define primitive physical events and then describe how, through an abstraction procedure, all primitive events can be defined.

6.3.1 Physical States of Objects

We first give an account of the meaning of the term "state." For us, the *state of an object* is the set of the spatial and causal relations involving the object. For convenience, it is also used to mean the deductive closure of the corresponding formulas of the set when doing reasoning about the object. Once again, we have to stress that the set of *all* relations will not do since it will contain abstract relations.

The physical state of a physical object, as a set of relations, includes:

- External interactions: Interactions with other objects,
- Internal interactions: Interactions among its parts.
- Spatial location.

For example, the state of an apple includes where it is, what supports it, its color, its weight etc. The first is its location, the second, an interaction. Although the rest are traditionally taken as properties, it is easy to fit them into our framework. Roughly speaking, color and weight both reflect the internal structure and organization of the object. Therefore they are internal interactions.

We do not include the past of an object in our notion of state. This is consistent with the usual reductionist view on physical world, i.e., the current state bears the consequence of any event in the past and is therefore sufficient to uniquely identify the object and predict its future.

This view, of course, is not quite solid. For many objects we do not know the effect of certain events on their states. The only thing we can do is to explain a happening by past events. In particular, people change their mental states and we know that these states, although not directly observable, have a direct impact on their behaviors. As a partial remedy, we introduce some special kinds of events to deal with them. For these kinds of events, the operationality is enforced in terms of regular relations between current behaviors and past behaviors.

6.3.2 Physical Events as Physical State Changes

Physical events can be simply defined as the changes of physical states. The states of objects are changed if and only if there are some interactions ceasing to exist and some other interactions beginning to exist. Namely, if we have an event E involving a set S of objects then there should be some states of some x's in S that are changed.

Note that up until now we have been talking about interaction changes without any reference to the interval of observations. Since the change of interactions proceeds in very small time intervals, we observe the event as an almost continuous picture. This is how people get different impressions between running a distance as opposed to walking a distance.

If we were to look at the process at the beginning and at the end, we would not see the difference. Recalling the model mapping picture, we are now on the plane of observational model. This is how we form the notion of identities of individual events. However, we will soon deviate from this track, to make concessions to the need to make event descriptions tractable.

Based on this definition of events, we introduce the function "Participants." Participants(E) of event E are participants of interactions that undergo changes *during* the events. This notion of participants does not care whether, at the end, the state of an object is changed. As a matter of fact, when an agent participates in an event, the agent often, while doing something in the process, changes another object but itself returns to the original state. In the future, we may occasionaly use the function "participants" on a relation if the relation is part of the state of an object.

6.3.3 Types of Primitive **Physical Events**

We can classify events in terms of the states (or interactions) they change. There are four basic types of events (or state changes) in our model. They are:

1. Perceiving: E(A, x) where agent(A);
2. Moving: E(x, positionl, position2);
3. Internal change:
   (1) P(x)LE(x)LQ(x) (Property change);
   (2) E(new x) (creation);
   (3) E(nil x) (dissapearance);
4. External change:

   Let E≡E(x, y, z), I and I' be types of interactions,

   $$I(x,y) \prec_! E \prec_! I'(x,y) \vee I(x,y) \prec_! E \prec_! I(x,z)$$

Let us remind the reader that in our notation, E(x) docs not indicate that there is only one argument or participant in the event E. To know the exact number of participants, one should apply the function "Participants" to E. E has a single participant if and only if Cardinality(Participants(E))=1.

In the above types, except for perceiving, an agent can cause (i.e., initiate or force) the event and make it an action.

A primitive physical event can be of any of the above types or a combination of several of them *as long as the participants of the interactions are the same.* As a matter of fact, it is often the case that a higher numbered event type subsumes a lower numbered one. For example, an external interaction is often accompanied by a location change.

Now we briefly explain each of the basic types.

1. Perceiving

   Perception, being the type of events only initiated by agents, has been discussed in the section on agents. We recall (cf. the section on agents) that, here, the state change is beyond direct observations. But we operationalize the consequence of the event by requiring that the event generate an information object or be followed by a relevant action.

2. Moving

   Moving is location change. For the event E(x, pos1, pos2),
   $\mathrm{at}(x, \mathrm{pos1}) \prec_! E \prec_! \mathrm{at}(x, \mathrm{pos2})$.

3. Internal change

   Formally we have for (1)
   $$\exists P,Q\ [P(x) \wedge \neg Q(x)] \prec_! E \prec_! [\neg P(x) \wedge Q(x)], \tag{5.1}$$
   for (2)
   $$E \Rightarrow \mathrm{created}(x), \tag{5.2}$$
   and for (3)
   $$E \Rightarrow \mathrm{nullified}(x). \tag{5.3}$$

   The internal change is the result of the interactions among parts of the same object. (The object may not be actually viewed as a composed object at the level at which we observe it.) A change of the internal states of an object may only manifest itself as a property change. But it is also possible that the change makes the object no longer an object accepted by our mapping procedure, i.e., the event makes it leave the abstractional model. In this sense it disappears. Conversely, some objects may be "created" or introduced into the model.

   The three subtypes of events can be combined. For example, if cooking materials are considered part of the system then the cooking will make them disappear but create a new object: food.

We caution that not all unary relations are *"inherently"* internal states. For example, age of a person is a historical relation, not a state. The address of a person tells us his external interaction rather than anything internal. For another example, although some unary relations do reflect the inherent properties of an object, one can still relate it to some external conditions: the color of an apple makes sense only under normal lighting, the weight of a person depends on the gravitation acceleration of where he stands, and so on.

We note the inability to recognize the category of a relation (e.g., causal, or social, or historical etc.) from its logical form is a substantial problem in making use of many of the constraints we have. But this is the problem of a particular representation system, not the conceptualization itself.

4. External change

   The formal definition of external change indicates that there are two main possibilities. First, the interaction between two objects may change into another kind of interaction. Secondly, one object x may stop interacting with another object y, but start engaging in an interaction with a third object z. In the latter case, not that we do not care about the fate of y but that if it "falls into" interaction with something else, it can then be characterized as a participant of another event.

Our claim is that *all* physical events in our operational model can be incorporated into these basic types and combinations.

We now discuss some constraints on physical events. First, because the definitions are completely based on interactions, no abstract objects are involved. Secondly, it is absurd to have E(x, x) for any x since an object does not interact with itself. As a matter of fact, all the axioms restricting interactions can be "transplanted" here. We will not elaborate further. However, the above definitions introduce more constraints. For example, when an object x is the only participant in event E, according to axioms on internal events, it is impossible for x not to change if it does not disappear and nothing else gets generated. However, these constraints tend to be obliterated by our actual representation system which does not make as sharp distinctions. As a result, we cannot make use of them.

6.3.4 Event Abstractions

The mappings from events in the observational model to events in the abstractional model should be used to enrich our vocabulary if we would like to talk about things at many abstraction levels [Davidson 1980]. For example, we would like to see some physical events in terms of their social functions. For instance, we prefer to say that a person bought a house instead of saying that he signed a contract with his name on it and according to the contract he can live in the house and so on and so forth.

In this subsection, we generalize the notion of primitive events to enable us to say more pertinent things. We note that this is strictly semantical. In addition to events involving only direct interactions, we may view an event sequence that is causally related (i.e., on a causal chain) as a primitive event in our model. The condition is that not all the intermediate interactions and interacting objects be present in our model because, in that case, the event should be composed instead of being primitive.[1]

Next we allow certain abstractions of physical events to be primitive. There are two observations about people's "feeling" for abstractions: (a). Abstraction relations are more comfortable for us than other relations. E.g., if "CoverOf" is a relation (function), and "Content" an abstract relation, it is more acceptable to have "perceive(Content(a bookcopy))" abstracted to "read(bookcopy)" than to have "takeOff(coverOf(a bookcopy))" composed into "undress(a bookcopy)." (b). Certain abstractions are more comfortable for us than others. There should be good reasons for those particular abstract relations to be favored. But we will not get into it now.[2]

The event abstractions can be of three kinds:

1. Causational abstraction.

   This abstraction procedure will take a sequence of events that are in the same causal chain and make it a single primitive event in the abstractional model. For example, in a prototypical shooting, one pulls the trigger, the bullet flies and hits someone, and someone dies. All this can be represented at the abstractional level by one symbol "shoot" and the consequence could be "dead."

   Formally, we say there exists an event e(x,y) and e⇒p(x, y) for x and y, if

$$\begin{aligned}&\exists E,n[E\equiv[e_1;e_2;\ldots e_n] \wedge E\Rightarrow p(x,y) \wedge\\ &\qquad \{x,y\}\cap \text{Participants}(e_1)\neq\emptyset\wedge\\ &\qquad \{x,y\}\cap \text{Participants}(e_n)\neq\emptyset\wedge\\ &\qquad e_n\prec p(x,y)].\end{aligned}$$

[1] We can get the same brevity by defining a composed event but that is a separate problem.

[2] We trust that the reader can come up with weird abstractions as we see in the chapter on mapping.

Note that the clause e_n-<p(x,y) is necessary, because p(x,y) must be true after the last event, but it need not be the effect of the last event. We also note that p(x,y) may only contain x or y.

2. Participant abstraction

In this case, the abstraction of an event is based on the abstraction of the object that is involved. Typically, the physical form of an information object can be abstracted to the symbols it designates or the content it expresses. For example, one may prefer to say

Read(A, a file)

rather than to say A is looking at the display of a file or a hard copy of a file. This can happen to aggregation, too. For example, we may say

deliver(a mailman, a collection of mail, a block),

to mean the abstraction that he delivers mail to every house in the block.

In the case of abstracting from the physical forms of information objects to symbols, the rules are as follows. •

Let a symbol *9^* and a physical form *9$* be in the relation

3! fy[designates(fy, *$+)].*

(Note that this relation does not need hold for all of our observations, it suffices that it is true for only the model we care to have.) We will have

$\mathbf{Location}(\theta_\psi)=\mathbf{Location}(\theta_\phi)$,

i.e., moving *9$* would result in moving *9^*. Also, any spatial relation *9$* is in can be transferred to a relation involving *9^*. Also, we have

$\mathbf{created}(\theta_\psi)\equiv\mathbf{created}(\theta_\phi)$,

and similarly,

$\mathbf{nullified}(\theta_\psi)\equiv\mathbf{nullified}(\theta_\phi)$[3].

Some mapping in terms of equivalence would narrow down the consequence of an event. For example, the implications of perceive(A, *9$)* and perceive(A, *9^)* for agent A intersect. The former includes where *9^* is and how much it weighs.[4]

[3]The predicates "created" and "nullified" will be defined in the chapter on systems.

[4]On the other hand, in general, the agent A need not necessarily recognize *0$* as designating *0^*. However, in our operational model, we have made the simplifying assumption that a symbol will be recognized by all agents of the same kind.

We caution that this type of abstraction is limited. Sometimes we cannot directly abstract a physical event involving physical forms of information in this way. For example, a hardcopy of a book can become wet or a disk file can become damaged. In both cases, the physical objects are changed but it is difficult to find a proper mapping that reflects what happens to their corresponding abstract forms.

We, in event specifications, talk mostly about information objects at the level of physical form or abstract form (symbol). We, in general, do not abstract event participants to the level of concept (abstract content of information objects). This is because at the level of abstract forms, the objects are still targets of interpretation and, therefore, some operations can be viewed as happening to them. However, it is hard to say how one can do anything with a concept. (Although for almost everything we do, we rely on concepts in our minds.)

3. The combination of causal and semantical abstraction

This combination is interesting because it is in this form that events on social relations are abstracted.

The abstraction rule is stated as follows (logical symbols would be too messy):

If an observed event E involves an agent A and a symbol θ_{ψ}, which is a definite reference of an object w, and the consequences of E are a social relation between A and w, then E can be abstracted as involving w.

We also note here that, from past discussions, the relations between symbols and physical objects are limited to spatial relations when there is no agent present. Symbols only interact with agents. No concept participates in any interaction.

As an example, we explain how we allow a statement such as

    buy(A, a company)

to be a primitive entity in our abstractional model.

Buying a company is not like buying an apple. The commodity does not change hands. It does not even change the commodity, i.e., the company, in any physical way. What happens is that some statements are made and some documents are prepared and signed. *Because of these physical changes* the social relations involving the company change. The ownership of the company changes. We notice that three things have happened to our observation. First, the words on the documents are recognized as symbols standing for the company, the buyers, and the sellers. Secondly, the process of finishing all the paperwork is distilled into the final consequence for the parties of interest. Thirdly, the final consequences of physical occurrences are interpreted in terms of what the symbols stand for, namely the social relations involving the company and the people.

Note that with the possibility of talking about things in a more abstract way, we lose many constraints we used to have when we had only physical events. For example, at that time, we could require that all the participants be spatially close, since they were to interact. However, since an object can be represented by a symbol in an event, the constraint no longer exists. On the other hand, there remain other constraints. For example, usually the agents exist spatially close to the location of events. Also, the consequences would be relations involving the realizations of those symbols (or the newly permitted "abstract" participants).

Having worked this out, we can see that even events such as sell(A, Patent), or sell(A, CopyRight(a poem)) for an agent A and abstract objects Patent or CopyRight, can find a mapping and become primitive events.[5]

Preconditions and consequences

This abstraction mechanism for events reconfirms the usual way of describing primitive events at the level of abstractional model, namely, using preconditions and consequences to characterize events.

Formally, relation P is a *precondition* of event E (or P *enables* E), denoted as P»-+E or PG Precondition(E), if

[-.P-+-.E] A [P -<tE] (6.11)

P is a *consequence* of E if

E -*P(| E|) (6.12)

or, more exactly,[6]

$$E \Rightarrow P.$$

While the same characterization can be used for both models (observational and abstractional), we point out that the uses are conceptually different. In the observational model, we talk about events in terms of beginning and ending points while our observation spans the whole duration. This duration may contain many time quantums and our observation includes the object states for each of them. On the other hand, in the abstractional model,

[5]But in this case, we would be dealing with abstract contents of information objects, which our theory is not quite prepared **for.**

[6]Note that (C.12) is a temporal **relation.**

the observations at the beginning and ending points may be the only information source we have. The states at the two points may be equated to the event "per se."

If we remember that the consequence of an event is not the event per se, then the events in the abstractional model are not self-explanatory. For example, only when resorting to the observational model, one can justify saying that running in a circle is an event.

Finally we introduce some definitions based on the notions of preconditions and consequences of primitive events. They will be used frequently in the future.

Event E enables event E' if

$$\text{consequence(E)} \in \text{precondition(E')}; \tag{6.13}$$

Event E (or relation R) is said to *disable* event E' if the consequence of E or R implies the negation of one or more clauses in the precondition of E'.

Event E(u) is said to value-affect event E'(v) where u has an attribute x and v has an attribute y, if (a). E happens before E', (b). u.x changes values during E, and (c). the value of v.y is related to the value u.x by an equation.

In the above three cases (enable, disable, and value-affect), event E is said to *affect* event E'.

The predicate "affect" can be generalized. If we denote the original "affect" as "directAffect," we can say that event(or relation) E affects event(or relation) E' if

$$\text{directAffect(E,E')} \vee \exists \text{E"} \text{ [directAffect(E,E")} \wedge \text{affect(E",E')]}. \tag{6.14}$$

A set of predicates S affects another set S' if there is at least one predicate in S which affects a predicate in S'.

6.4 Object Composition and Identity

The problem of object identity presents a great intellectual challenge to our reflective thinking. Questions are asked: If a facade of a house is remodeled, is the house a "new" house or is it just a changed "old" house? If a person is changing his bodily constituents by a large percentage, daily and monthly, why do we always take him/her as the same person? In this section, we will briefly review the issues.

The identity problem arises out of the existence of composed objects. If all objects were atomic, we would not have the problem. Everything would be itself and nothing else. It would be also nonsensible for it to change into something else because it then would not be atomic. So the discussion of identity is inevitably intertwined with the definition of composed objects.

On the other hand, composed objects and their likes have some interesting properties in their own right.

6.4.1 Composition and Collection Objects

A set of objects can form a structure as loose as the sand on a beach or as tight as an organism like the human body. For the convenience of reference, these structures are called "collections," "compositions," "collective objects," etc., which mean different things to different people. There is no point in arguing which term is right for which kind of structure.

In the context of this thesis, a composition object will be viewed as having a fixed structure, the constituents of which stay in close connection, and, therefore, a "real" object. A special subset of composition objects is the set of systems. The notion of systems **will** be analyzed in detail later. At the other extreme, collections are those sets of things that only exist at the same time and do not affect each other in any causal manner. This is a useful approximation but apparently an exaggeration, too. Human bodies, well organized as they are, constantly assimilate things that are alien, which have not had any connection with the bodies before. On the other hand, books on one shelf or sands on one beach may engage in mechanical interactions.

In general, we define a physical composition object S as *a set of physical objects* engaged in certain *regular interactions.*

There is a certain number of objects £ and a certain number of interactions 9 that are *definitive* for the composition. If any one of them *ceases to exist,* the composition as a whole does, too. There is a subset of S and a subset of 9 that are *identifying.* If any one of them is *changed* into a different object, then the composition S as a whole will, too.

The exact *definitive sets* and *identifying sets* are specific with an individual composition.

The above definition can successfully explain most intuitions about object identity and is consistent with our theoretical framework.

We note that these definitions are very close to the notions of "structure" and "organization" of Maturana's [Maturana and Varela]. For Maturana, an organization is the set of (the classes of—K.Y.) "relations that define a machine as a unity, and determine the dynamics of interactions and transformations which it may undergo as such a unity." And the structure of a system is the actual relations that hold among the actual components. The change of identity is further explained in terms of the change of classification of an object (or a system) according to specific organizations. This way of approaching the issue of identity is profound but not as general as one wishes. It is profound because it points to the way people usually identify things, i.e., things are identified according to their classifications. If something is to be classified differently because the changes it undergoes then we may consider its identity has changed. This clearly reveals that the notion of identity is relative to human cognitive activity. It also explains our intuition. For example, if a dead person is not identified as the same person that lived, it is because being alive is a criterion in classifying things. On the other hand, our definition has avoided the notion of classification because we feel the notion of identity is ontologically prior to the notion of classification. Namely, we accept many things as behaving in somewhat regular way and *different from anything we know*, i.e., accept their identities, before we ever try to classify them.

A collection object, on the other hand, is simply a set of objects which may engage in interactions that are not regular.

We do not define "identity" and "existence" for collections. This is partially due to conceptual reasons and partially due to the fact that they easily group and regroup. We may refer to a collection as if it were a single object if it can be treated that way. The checking of its nullification is reduced to the checking of the nullification of all its constituent elements.

There is a set of collection objects that is of special interest to us. These are collections that are of some common or related use to us, people. In particular, they may be physical forms of information objects. We note that the lack of organization in collections is often eclipsed by the use we assign to them. For example, a pile of papers can be viewed as a whole if they are to be used together for some occasion. The dishes served to a customer are thought of as a whole because their use is closely related.

In particular, the use may be physical forms of information. In that case, the "structuredness" is even stronger. For example, a deck of cards and a pile of blank papers share similar physical status. However, the deck of cards seems much more a "composition" to us than the pile of paper. We point out that this is because there is an *abstract composition object* which is the content behind the physical forms. However, that is beyond the scope of this thesis and we will stop here.

6.4.2 Events to Composed Objects

We use the term "composed objects" to mean both compositions and collections.

We note an event involving a composed object can relate to its members in the following ways:

1. Distributable

   A distributable event is equivalent to the same event happening to each individual.

E.g., walkTo(Jones', theater) is presumably the same as for every x in Jones' family: walkTo(x, theater). It is especially in cases like this that collections are used, since it is convenient to talk in terms of the whole.

The basis for this, we note, is that many properties of a composed object are distributable to its parts. This is particularly true with spatial relations. In this case,
$E(S) \equiv \forall x \in S\ E(x)$.

2. Distributable in consequence

   There exist corresponding events for the individuals but the event happening to the whole is not equivalent to the set of events (that is necessary for the effect) happening to the individuals. E.g., when one brings a collection of letters to a family, this is a single event. It is not the same as when one brings one letter at a time to the family. Although in certain contexts, the consequence concerning us is the same.

3. Internal events

   The internal event has no direct consequence on each member. As a matter of fact, we cannot even "find" the types of events to describe what happens to the members. E.g., a deck of cards gets shuffled, a group of people quarrel. Both the event and the consequence can only be talked in terms of the whole. This case is the same as the internal events we introduced as a primitive event type. Here the interactions among the parts become more discernible.

Chapter 7

Operational Models of Systems

7.1 Basic Concepts of Systems

7.1.1 A Definition

A system can be viewed as a composed object, the constituents of which behave in a certain consistent manner such that the object as a whole exhibits some regular behavior. From the above discussion, we know that that manner should be a causally related manner. From a behavioral viewpoint, a system can be identified with a process, which involves objects, each playing a certain role for the duration of the process.

Semantically, system models are products of mappings. And they are the most conspicuous examples of the mapping process. Every system description selects (through the mapping) a set of primitives to build the model upon. Particularly, for every system there is a set of event types that are selected as primitives. Any observed instance of one of the types is to be recorded in the specification of the system.

Formally, a system S can be defined as a 7-tuple

$$S=(E, A, T, R, e, H, H)$$

where S is the system's components, A the noncomponents, F the goals (class), R the relations (class), *e* the events (class), S the time-independent constraints (i.e., axioms in terms of the logical representations of the constraints), and H the history of the system. They are defined and explained in next subsections.

For brevity, we may use the same symbols for a function that maps the system S to the corresponding element in the tuple. For example, for the system S

$$E(S)=E.$$

As another notational convenience, we use |S| to denote the constituents of the system S. If we have a primitive function ConstituentsOf then

$$|S| = \text{ConstituentsOf}(S).$$

|S| means all the objects that have ever existed in S. We use $|\varsigma(t)|$ to denote *all objects existing in S at a given moment t.*

Obviously,

$$|S| \supset |\varsigma(t)|.$$

In the future we may simply use S(t) if no confusion will arise.

7.1.2 Components, Noncomponents and Surroundings

There usually are two understandings of the term "system." In one sense, "system" includes both the system "per se" and the part of the environment with which the system "per se" interacts. In another sense, the part of the environment is not taken as a part of the system. We have adopted the former sense in this thesis. For example, if we are talking about pumping water using a pump, both the pump and the water would be parts of the system *where* the water gets pumped.

But we share with the latter the same intuition. Objects in a system are further divided into components and noncomponents. In the pumping case, the pump is a component and the water is noncomponent. They both belong to |S|.

Everything else that is not in |S| belongs to *surroundings.*

We first discuss briefly the criterion as to when $w \in |\varsigma|$ for an object w, i.e., when w is a part of the system or, put in another way, when w is separated from the surroundings.

This amounts to drawing a boundary between what we would describe and what we would not describe. This boundary defines the interface between the system proper and its *surroundings* and distinguishes one from another.

This boundary can not simply be a spatial one. For example, we may come to a restaurant to do business although most commercial transactions take place in offices.

If we can decide which objects are components (this should not be too hard as we will see), a more sensible way is to use the interactions with components as boundaries. For example, an object is considered to have entered the system S the first time there exists a relation

$$r(w,\sigma)$$

and $\sigma \in \Sigma(S)$. It is considered to have left the system after the last relation $r'(w, \sigma')$. But this would not always work. A noncomponent can interact with another noncomponent to induce some change to S, even after its last interaction with a component.

This suggests that there cannot exist any criterion based on ontological principles alone. A viable scheme for deciding on this is to resort to the model mapping scheme we described earlier. There we specifically mentioned causal relations as the criteria for a completeness definition of system descriptions. Given the supposed goal states or events, causation would be a criterion for the selection of facts. However, which situations are taken as goals is completely up to the observer.

We note here that, since, eventually, the system is the only thing resulting from mapping, to say something exists in the system is equivalent to saying that it exists in the world we care to observe or simply saying that it exists. I.e., in our specifications

-oc e|S|=-»exist(x)

or, equivalently,

x €|S|=exist(x).

We may use "being introduced" instead of "being created" to indicate that an object starts to exist. This is just a choice of words. Both mean that the object being referred to is not identical to any object currently existing in the system and not identifiable with any objects that ever existed in the system. It should be considered "new."

We should caution, however, that, *in general*, being observable for the duration of a system is not equivalent to being observable at a particular moment. Therefore, it is not always true that

hxE|S(t)|]=-iexist(x). •

In the future, we will see that, for a class of systems this is true and that many powerful constraints will follow from it.

Conceptually, *components* are understood as the inherently internal structure of a system, part of the underlying mechanism of the system "per se," and existing in S permanently. So,

Vw[w€E=Vt we|c(t)|].

On the other hand, for noncomponents A,

Vw[weA=-.Vt w€|c(t)|]-

Obviously,

|S| = E U A.

We have defined components purely in terms of temporal durability. This is partially because the decision procedure would then become very simple, and partially because we do not yet have a better alternative.

It is more likely that an object is a component if it *regularly interacts with objects of the same class in the same manner rather than with objects of different classes in different manners.* Consequently we perceive it to be "part of the mechanism" and emphasize its permanence.

A system is *closed* in the usual sense if A is empty. The inverse is not true since a system can still exchange interactions with its surroundings. In this research, only open systems are of interest to us.

7.1.3 The Events and Relations in a System

e stands for the set of 3-tuples for each event class: the event, the precondition of the event, and the consequence of the event (cf. the chapter on formalizing domain categories). As a notations! convenience, we use *\e* to denote the set of event classes in *e.*

We recall that (cf. the chapter on formalizing domain categories) events and relations can stand in interrelations that are interesting but other than directly causal. Event A (or relation A') is said to *enable* event B if the consequence of A or A' implies one or more clauses in the precondition of B. Event A (or relation A') is said to disable event B if the consequence of A or A' implies the negation of one or more clauses in the precondition of B. Event A may also value affect event B. In the above three cases, event A is defined as *affecting* event B.

A group of enabled events, when not value-affecting or disabling each other, can happen concurrently and result in the same system state.

We now introduce the following two definitions. They are useful in conceptualizing on events in a system.

An event *path* is a set of events related by "affect" relations.

An *independent path* is a transitive closure of the events under the "affect" relation. Independent paths can be simulated separately for a system description (cf. the chapter on simulation).

H, the history, is a large and complex composed event class involving all objects in S. Its duration is the existence of the system. A composed event will contain relations, so H has or implies all the system states.

We note that this definition is conceptually a bit awkward. We could have defined the history as an event instance that was a unique individual entity, and then generalized it to event class. Theoreticians would feel better that way. Our treatment is just a shortcut to make the presentation simpler.

This event H is rather complex. In particular, the occurrence of many events will be contingent upon the situations at a given moment. In terms of its description, there will be conditional events or other indeterminate structures. For all the alternative subclasses of events, only one will be actually observed. These subclasses of events are called *traces.* A trace does not have alternatives. It is an event sequence (class). Relative to the history, we may say that it is an "individual run" of the history, although it is still an event class.

The notion of trace will be explained further in the chapter on simulation. Now it suffices to note that it, as an event class (or event), has part of |S| as participants. I.e., if T is a trace of S,

$$\text{Participants}(T) \subseteq |S|.$$

Now the set of possible traces (classes) that are observable for the system S would be the set counterpart to the history (as event class). If we introduce a function TracesOf, meaning "the traces of," we will have

$$\text{TracesOf}(H) = \{T_i | \ i \ \}.$$

R and B, the classes of relations and the set of axioms, need little explanation.

By definition, all relations in B and H are members of R. All events in H are members *ofe.*

7.1.4 The Goal Structure of Systems

Systems are assumed to behave with a purpose. Within the scope of our research, this purpose is taken as a set of relations and events (types), possibly temporally constrained. Therefore, the logical form of a goal is a first order predicate formula. Some behavior patterns, say, remaining stable or continuing growth may also be thought of as goals. They are then meta-goals, which we will not discuss.

The goals in a general operational system are different from goals in normal AI problem solving systems, where a goal is meant to be the final state of the system. The goals here may be scattered at many time points in the system history.

Undoubtedly, goals are decided on the basis of the set of desiderata of the designers of the system, e.g., the security of a system. But our notion of goals is operationalized. That is, we do not talk about what is on people's minds. The only things that concerns us are the observables.

Because our model is concerned with qualitative states, we do not investigate performance goals unless the performance differences are embodied in different states. For example, an editor can implement a command "kill region" which achieves the same effect as a series of deletions. In that case, both the two goals have to be put into the goal set separately and explicitly.

A system often has many goals (as defined and restricted above). The goals are categorized by their associated agents (e.g., in a typical purchasing situation, the goals are: the buying side gets the goods, the selling party gets the payment), temporal constraints (e.g., two relations may need to hold at the same time or need to not hold at the same time), and causal chains for achieving them (e.g., some need a long chains of events to be enabled, some do not need any prior enabling events. In the case of a restaurant, a lot of things happen that do not have any direct bearing on how the customer gets to eat the food, but they are still a necessary part of the operations in a restaurant, e.g., the customers are seated.)

Major Goals and Minor Goals

We elaborate on this last categorization a little bit. Goals in a system can be divided into two groups: major goals and minor goals. For a major goal, there is a chain of "affect" relations of length greater than 1. For a minor goal, the length is 1 or 0. A goal may lie on the chain for achieving another goal. But we may omit it in specification if it is apparently redundant. An example of this would be, if we know it to be causally or logically equivalent to another goal.

This is especially true of social processes. For example, in a restaurant, the customer is to be seated. This should be one of the *requirements* of the restaurant, but it nevertheless does not have as much constraining power as the requirement that a customer is to be served with food. This is because if the customer is inside and there is an empty seat he can be seated (Presuming this is in a restaurant of modest size, we can imagine reservations, ushering and other fancy things, of course). On the other hand, much communication and cooperation is necessary for a customer to eat a special type of food. Seating is a goal, still, because without this the customer would not consider himself adequately served just as if he were not to get food or drink.

When we say the goal of a system is a set, we have both groups of goals in mind.

Minor Goals as the Performance Requirement of Major Goals

The argument may go further than we want. One may maintain that almost any event has a minor goal in itself independent of what the ultimate objective is.

This is particularly obvious with physical processes. In the physical world, similarly we have certain ways of doing things or implementing things. The means of achieving or the "implementation" can be readily justified or rationalized. We can see the "secondary goal" clearly.

For example, in some planning programs involving individual human activities, although alternative operators are explicitly means rather than ends, the choices made imply secondary considerations which are really performance goals. E.g. when planning to go to

Boston from Stanford, one takes an airplane rather than walks. This is based on speed considerations. When one takes a bus instead of taxi it is because of expense considerations.

This seems to be true with social processes, too. For example, even ordering dishes could be thought of as more than a prerequisite to cooking the dishes, because we could have had a different tradition of going into the kitchen and picking whatever we want ourselves. Namely, in general, ordering is not really causally necessary in getting food in a restaurant: It is possible that looking at the menu and speaking to someone may be a real source of pleasure and part of the goal. It is the particular culture that requires us to do it in a certain way: Sit there and order. This is not absolutely physically necessary, but socially necessary nonetheless.

The problem is that, here, the causal chain associated with the minor goal is less easy to identify. If the requirement of seating can be completely attributed to necessity for adequate digestion, the role of seating would be much better understood. However, because it is difficult to define what underlying reasons exist for us to prefer sitting around a table over other alternatives (when dining), seating remains a requirement.

This problem is evidential in other works, too. In story understanding programs those "required" activities such as seating are only part of a "script." Their role is just to help to identify a situation. There is no argument justifying or even speculating about their existence. Their role in real situations (rather than in the processing program) is left unexplained. (We are not talking about case by case explanations. Of course, sitting is more comfortable than standing. But the question is why people are seated in a restaurant.)

The solution to this problem is our pragmatic approach. In the first place, our mapping procedure allows us to simply not look at anything that is not of interest to us, whether we can see it or not. Furthermore, our practical interest in making distinctions between goals and non-goals is being able to do checkings based on teleological constraints. If an event or relation is on a causal chain, then it is causally necessary for a goal to be achieved in this particular way, i.e., through this particular event. We could just view this particular side as specific implementation and ignore its minor goal aspect (if any) as far as obtaining the major goal is concerned. For example, we do not view giving an order as a goal of a restaurant, but since the chef needs orders to cook, the event of giving an order will be checked in some way.

7.1.5 Remarks on the Definition

We note that the above definition of systems cannot mimic the definition of a finite state machine, which introduces state transitions instead of a history. One of the obvious reasons is, of course, there are an infinite number of states, since the number of noncomponents are infinite. But that is not essential, a proper abstraction can handle this. The reason is more technical. That is, the number of states and transitions is relatively large and the number of transitions per state is fairly small, in many cases the transitions are unique.

It is important to note that the elements in the tuple, Γ, R, ε are all classes rather than instances. We could have made them instances, but that would ruin our plan to make the whole definition a definition for *classes* of systems. In fact, given our current way of defining systems, simply abstracting objects in $|S|$ would achieve this objective. However, there is a slight problem. The definition now leaves the behavior of the system ambiguous. This is because if there are two objects of the same class, then there can be two possible assignments of their roles in individual events in the history. Nonetheless, the problem is not serious. When the two objects become indistinguishable it should be the case that it is not causally necessary that they be distinguished. We, therefore, need not worry about it.

For example, in a statically stable system (see a later subsection for details) one may map the same individual observed on a larger scale (e.g., a longer interval of time) to different or unidentifiable individuals. E.g., in the restaurant example, we may view the customer who comes back the second time as someone new. As long as the operational consequence is the same, there should be no substantial problem with this mapping. On the other hand, this treatment will entail interesting constraints for the semantic analyzer to explore.

Lastly, some words about the "control" of a system are in order. This is because, whenever there is a system, the first thing we, in computer science, look for is the "control."

It should be noted that normally, the word "control" would come up only in reference to external manipulations of a system. In our system, the behaviors of objects are results of their interactions. No external control exists. The "control" or the achievement of regular behavior is exercised by agents or components that initiate agentive events. (cf. the section on agents in the last chapter) An example of the latter may be the governor in an improved steam engine. An agent, in turn, makes decisions on five kinds of occasions: a. when the situation results in a social relation that necessitates his action b. when he acts based on previous knowledge c. when he acts based on the information about the situation he directly perceives. d. when he acts based on the information conveyed by information objects. e. when he acts based on the combinations of all of the above. The event types corresponding to all these occasions were discussed earlier.

7.2 Stable Operational Systems

7.2.1 Behavioral Stability

Stable systems exhibit recurring behaviors. We formalize as follows:

Stability Assumption:

if stable(S) then[$\exists$e H(S) $\equiv$ repeat e], (4.2)

where e, as pointed out earlier is an event type. Every instance of e is called a "recurrence."

A stable system may not stop or restart *right away* when all the goals have been achieved. Some events should happen to take care of changed component states to guarantee recurring behaviors.

7.2.2 Structural Stability

The set of components of stable systems is *defined* as

Let H(S)$\equiv e_1;e_2;\ldots e_n$,

$\Sigma(S) = \bigcap_i$ Participants(e_i)

The rest of the objects involved in e_i are noncomponents.

$\Delta(S) = |S| - \Sigma(S)$.

We denote components in a recurrence e_i, Σ_i. Obviously,

$\Sigma = \bigcup_i \Sigma_i$.

This definition of the components of a stable system cannot be proved from the definition of components of general systems. The more general definition only requires that for a component x,

$\neg\exists n\ \forall i[i>n \rightarrow \neg[x \in |S_i|]]$.

This means we can have cases where a component appears in an infinite number of recurrences of a system but not necessarily in every recurrence. Our new definition will avoid that.

This treatment is more appropriate and more tractable. It is a useful abstraction to work with. E.g., if you have replacement players in a basketball team, then, according to the general definition, they are components which do not appear in each game. Now if you are just interested in the most essential part of the sport, e.g., the rules of the game, or the behavior of a team as a whole, you may not be interested in exactly who plays what. This means you could view the team as if it did not have replacement players and every player appeared in every game. However, if you would like to observe the strategy of playing, or the performance of the team in a season, you may then want to include the replacement players. This time, however, you will naturally have to extend the span of the system cycle, i.e., the duration of the recurrence. In that case, all the players will appear with their identities, and the system components will still meet our new definition.

While a component is "required" to exist for each recurrence of a stable system, a noncomponent, in genéral, does not necessarily exist for exactly one recurrence. For example, in a hospital, the medications and patients are noncomponents. They come and go at no fixed time. If we take daily activity in a hospital as the target of our description, the lifespan of these noncomponents may be longer than the duration of that recurrence. However, there might exist a collection of patterns of relations among system objects which stay on, although the identities of some of the participants of the relations, i.e., the identities of noncomponents, are changing.

Phenomena like this are so called *"dynamic* equilibria" in the physical world. In our stable systems, we can borrow the term to divide systems into two classes: dynamically stable and statically stable. In dynamically stable systems, the existence of an object can cross particular recurrences. In statically stable ones, all noncomponents will leave the system at the end of a recurrence.

Formally,

the noncomponents in a statically stable system follow

$$\forall i\ \Delta_i \cap \Delta_{i+1} = \emptyset.$$

Object Existence in Stable Systems

The general axioms on *object existence* take a special form for a stable system.

For components, the problem is simple. They would just exist. There is no creation, nor disappearance.

There are two cases with noncomponents. Some noncomponents are created at the time they are introduced into the system. For example, the chef's cooking of a dish creates it and introduces it into the operation of a restaurant. This is often true of composed objects. When composed, they come into existence. When decomposed, they disappear from the system and cease to exist simultaneously.

But some objects may be introduced into a system many times within different recurrences or even one recurrence. For example, a customer can patronize a restaurant more than once, even everyday, even many times a day.

If there were no latter case, we could have directly used the axioms and theorems for object existence in general. The checking rules for systems would be easily enriched in this way.

We observe we can achieve the effect by making a small revision in our notion of object existence for noncomponents. As a matter of fact, we play a trick similar to the trick we played with components. We will *assume* that the identities of noncomponents are changed each time an object disassociates itself from the system.

This notion of object creation apparently narrows down the ranges of systems we investigate. More accurately, it is an approximation of real systems.

In the restaurant example, this would mean that the manager should get new meat each day before the store opens, and a patron will not be recognized when he calls again. But the property of the stable system does not seem to change much. It is reasonable to assume, as a first order approximation, that the owner or waiter of the restaurant need not remember their customers. As long as customers pay for what they eat, the restaurant can continue its business. On the other hand, if one would like to describe how a frequent patron is treated, he can (and should) always extend the time span of a recurrence (e.g. more than one day), i.e., he will take another view of the restaurant operation, the duration of a recurrence would be months or years. ^,

Having accepted this assumption, the checking rules can now indeed be legitimately used. Because of this, we will use "create" and "introduce" interchangeably in future references. Similarly, "destroy" or "disappear" are interchangeably used.

We note that the above discussion is valid for both dynamically and statically stable systems; as long as an object is in the system its identity can not possibly change. It does not matter if the presence is across recurrences.

Therefore, for a stable system, at time t,

[-nxG|S(t)|]=-nexist(x)[1]

The axiom on noncomponents in statically stable systems (simplified):

$\forall\omega\exists i[\omega\in\Delta(S)\rightarrow$

[3T$_t$[T$_t$GH(S)A e$_t$e]|Ti||]A hcreated(o;)^$_!$e$_t$-<$_!$nullified(a;)]],

where e$_t$ is a particular recurrence, "nullified" means ceasing to exist in the system and is equivalent to the case when a nullifying event has happened. Similarly, "created" means an object has been created. Their exact meanings will be defined in a later subsection. This axiom states that for any noncomponent it is created during a recurrence e$_t$.

A corollary of this axiom is that at the end of each recurrence, no system state will involve any noncomponents:

Let Sfj$_{na}$i j be the final state of e$_t$,

VwG A(S) -n(3r:relation [r€ S$_{fina}$j jA we Participants(r)) ;

[1]Note that this is a time-dependent relation.

We have shown that the notion of statically stable operational systems is a close approximation to many real world systems. It entails many useful constraints for a checking program to explore. Since we will concentrate on this kind of systems, henceforth, unless otherwise stated, the term "system" will refer to statically stable operational systems.

Moreover, since there is no difference among the recurrences of a stable operational system, we will use the formula of a recurrence (class) for the system history. The notions of traces and paths will be correspondingly changed.

We finally note that we do not usually readily accept the same treatment of identity **for** components as for noncomponents. For example, we can treat the customers *as if* they had new identities each time they come but we cannot do the same for waiters. It is true that we can imagine a case where a restaurant is run by volunteers who come only once. But in this case, as the real identity is changed, we probably would like to know how the waiters get into the system. It seems we do not take the existence of a waiter as granted, whereas we take the coming of customers as granted.

This is an instance which reveals that permanence in a system is only a superficial criterion, (cf. the subsection on components) It has great heuristic value but is not a fundamental principle. We may say that it is a result of approximation.

7.2.3 Stability of Resource

A stable system often has some collection objects (abstract or concrete) whose changes of states recur. The recurrence of changes has a pattern such that some measurements of them go up and down but stay in a constant range. For example, the storage space and capital for a company buying and selling books would be such collection objects. These are, intuitively, resources. We call their measurements *capacity.* A capacity of a system will be denoted by A. The set of capacities in a system is defined as the capacity of the system, denoted by A.

In terms of types, a capacity is an abstract object composed with the type of the resource, a unit, and a number. It has to be an abstract object because the whole "resource object" may have to be "hypothetical," for example in the case of capital (money).

It is tempting to think that a capacity is always associated with a component. This is because a component is felt to be essential in the operation of a system and because the number of noncomponents is infinite. But we point out that, first, a capacity is defined in terms of the *measurement* of the *kind of objects* rather than the actual objects with same identities. Secondly, the resulting value of a measurement is relative to a particular time; therefore, the number of noncomponents to be measured cannot be infinite. As a result, the measurement for noncomponents would also be finite. On the other hand, we can safely assume that all the objects that correspond to or are accommodated by a capacity, e.g., book stock (vs. capital), are noncomponents.

The following axioms characterize capacities:

1. Measurement.

   $\forall \lambda \in \Lambda\ \exists w, f[w \in \Sigma \wedge \lambda = f(w)]$,

   where f is a measurement function. This states that a capacity is the measurement of another object, possibly physical object. E.g., we may say "the volume of a storage object is n cubic feet," where "n cubic feet" is the λ.

2. Compositeness of resource objects.

   At each moment, the resource is divided into two parts such that

   $\text{Opart}(\lambda, S, t) + \text{Apart}(\lambda, S, t) = \text{Constant}$,

   "Opart" and "Apart" (standing for occupied and available parts, respectively) are functions returning objects (abstract) of the same class as λ. When the object w for $\lambda = f(w)$ belongs to Σ, the values of functions Opart or Apart correspond to parts of w. Because of the linear relation between "Apart values" and "Opart values," we will mainly talk about "Apart values" and designate them by $\lambda(t)$. Intuitively, this is the available part of the resource at time t.

3. Resource correspondence.

   An occupied resource corresponds to some collection of noncomponents by some measurements.

   $\forall \lambda, t\ \exists f\ \exists w \in \Delta[f(w) = \text{Opart}(\lambda, S, t)\ ]$,

   where w is a collection object and f a measurement function. For example, the capital of a book business may be turned into the book stock through ordering and shipping.

   $\lambda(t)$ changes with events. If event E makes resource $\lambda(t)$ change to a new value, we denote the measurement of the difference between the consecutive values of the available part of a resource by $\Delta\lambda(t+|E|)$, $\Delta\lambda$ for short, i.e.,

   $E(t) \Rightarrow \exists \delta[\delta = \Delta\lambda]$

   where

   $\Delta\lambda = \lambda(t+|E|) - \lambda(t)$.

   Note that the symbol Δ in $\Delta\lambda$ is used as an operator not the symbol for noncomponents.

We use AA in two ways: a term and a predicate. As a predicate,

$AA \equiv \exists\delta[\delta = \Delta\lambda]$.

This makes it possible to simplify the above formula to

$E \Rightarrow \Delta\lambda$,

here time variables on both sides are implicit.

We note if e=> AA A e' =>AA' then we will have

[e & e'] =>AA"

where

AA" = AA+ AA'.

This can be proved from a property of the functions Apart and Opart. Since we did not introduce the property we may just take this as an axiom.

In general, the results in this section will mostly be simply stated. The reason is that although as qualitative criteria they are quite powerful in checking errors their quantitative counterparts are mostly straightforward. Some of them have been treated in the theory of operating systems.

4. Limit.

VA, t $3A_m[A_m \leq A(t) \leq A]$

Assuming A_m is the minimum of A defined in the above formula, an immediate corollary of this axiom is that for any event E(t) causing AA,

$[A_m \leq A(t) +$ AA$\leq A]$.

Assuming there are two events e and e' causing AA and AA' respectively; if both events happen we have the total change AA+ AA' as defined earlier. When the two individual changes are of the same sign, we would have a resource competition. To avoid a deadlock, a general theorem is in order:

$h[A_m \leq$ A(t)+AA+ AA'] V -i[A(t)+AA+ AA' $\leq$A]] ->
3e"[e" =>AA" A e ^ e N e'],

i.e., if these two events would result in violation of the upper or lower limit restriction, there should exist another event that compensates for the changes incurred by the first event accordingly.

There may be a causal chain such as

E⇒P ⇒E',

where E'=>AA and P and E involve the objects corresponding to A directly **and** indirectly. For example, E may be sending orders for new books by **a** book company to a publisher, while P may be the relation that the company owes the publisher **a** certain money amount. From the immediate corollary of the limit axiom we can infer that the possible enablement of E' will depend on the event E. In the example, if the ordered books cost more than the company can afford, E' can not happen.

In this case, either the system situation can be verified to guarantee E' will happen or some caution in ordering should be taken. Reflected in the description of a system, this is often a test in the form of a conditional.

5. Existence of a quantum size.

Assume there is a quantum size of the change of A, i.e.,

$3A_0VAA\ AA \geq A_0$.

The axioms on resource limit and quantum size of A change makes monotonic increments eventually leading to infinity if not bounded in some way. Therefore if the changes are to happen an infinite number of times as in a stable system, we have a theorem (Up and down):

3e[e =>AAA AA>0] A

3e'[e' =>AA' A AA'<0]

7.3 Deriving Constraints on Operational Systems

Many constraints on objects and events are inherent from the operational domain model. E.g., only an agent can perceive an object, in particular, an information object. However, there are constraints that can be derived from the basic causal and teleological constraints on the operational systems.

The primary goal of this derivation is to provide grounds for our various checking rules for the language. We stress that they should be a set of theorems provable from a small set of axioms. Otherwise, if they are stated as isolated facts, our model of the domain is likely to be an inconsistent one.

7.3.1 Causal and Teleological Constraints in Systems

The causal constraint on a system (or more accurately, a description of system) is simply that the system be operable. That is, the history of system S is enabled by an empty cause. This is stronger than saying that H(S) is provable. This because the latter would be possible as long as the history can be simulated based on the assertions present in a description. But some assertions made by a user may not be causally possible.

The causal constraints in a system are no different from those discussed in general.

The teleological constraints are special with operational systems. They are in the form of two basic axioms. The first (Completeness) says that all the goals should be achieved for all the traces in the history.

$$\forall T \in H(S)\ \forall g \in \Gamma(S)\ [T \rightarrow g].$$

It is important to note that both T and g are time-dependent. This is true for the next axiom, too.

The second axiom states effectiveness. Namely, all paths contribute to achieving goals.

$$\forall T \in H(S),\ path \in IndependentPaths(T) \exists g \in \Gamma(S)[path \rightarrow g]$$

From this we can directly derive the following theorem:

$$\forall T \in H(S),\ p \in T\ [p \in \Gamma(S) \vee \exists p'\ (p' \in \Gamma(S) \wedge affect(p, p'))]$$

A corollary of this is that if an object w is the only changed object in an event E and E is not a goal, then w should participate in at least one more event. Otherwise there will be no event enabled by E, which contradicts the above axiom.

Formally:

$$\forall E,w\ [E(w) \wedge changed(E, w)] \rightarrow \exists E'\ [E \prec E' \wedge w \in participants(E')], \tag{4.1}$$

where "changed" is a predicate, which will be true if and only if some property is changed by event E (cf. the section on event types).

We point out that a system goal itself can be taken as the highest level specification of the system. And this specification is legitimate since it satisfies all of the three basic constraints.

The constraints stated at the level of these two axioms subsume most constraints stated for individual types of systems. For example, the two famous principles for operating systems are no deadlock and no starvation which, respectively, require that a system be causally possible and that all goals be achieved.

The two teleological constraints can be readily translated into checking procedures, as will be explained in part II.

7.3.2 Axioms and Theorems on Object Existence

We define a predicate "exist" as follows:

$$exist(w) \equiv [\exists w'\ (w' = w)].$$

The equality here is interpreted as identity. The definition of identity is part of our abstraction procedure, not explicable in the logic formalism.

It is taken as corollary that

$\forall$w,p [p(w) $\rightarrow$exist(w)]; (4.3)

Our intuition on the existence of objects is reflected in the following axioms.

Let t,t_1,t_2 be time points, w an object.

Continuity axiom (assumption).

if [exist(w,t_1) $\wedge$ exist(w,t_2) $\wedge$ $t_1 < t < t_2$] then exist(w,t);

Creation axiom.

if [[$\neg$exist(w,t)] $\wedge$ exist(w,t+1)]

then [$\exists$e:event (endingTime(e)=t $\wedge$ creation(e)) $\wedge$ w$\in$participants(e)]

$\wedge$ created(w,t+1)

We note that a creation is defined as a type of event. Any event that meets this description would be an instance of it. This definition does not make any ontological commitments about what constitutes the creation of an object. That was delegated to the definition of identity in a system.

An axiom on nullifying of an object (the opposite of creation) can be similarly defined.

We can then show some theorems that arise from our intuition.

The fact that no event can happen to an non-existing object is a direct result of (4.3). We now prove an object cannot be created twice, which is a useful rule in checking a description.

Assuming otherwise, we have for an arbitrary w

($\exists$t,t_1:time)($\exists$e,e_1: event)

[e(w)!created(w,t);e_1(w)!create(w,t_1)] $\wedge$ $t_1 > t$;

We assume no intermediate nullifying events for w exist. (It can be proved similarly to be impossible.) This implies

created(w,t) $\rightarrow$exist(w,t) ;

Since there is no nullify(w) in the trace of w until t_1, we have

exist(w,$t_1 - 1$);

However

created(w,t_1) $\rightarrow\neg$exist(w,$t_1 - 1$).

Contradiction.

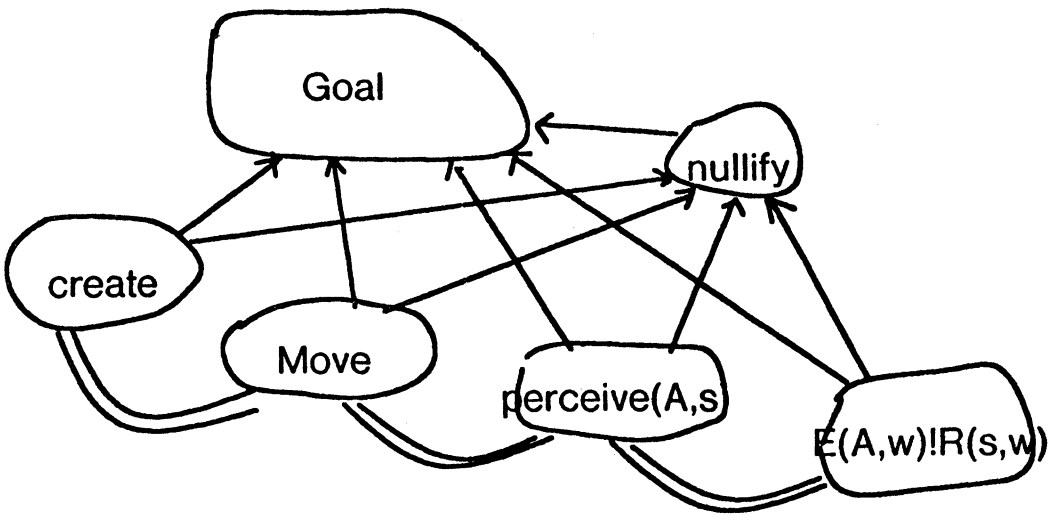


Figure 7.1: Possible Traces of a symbol

7.3.3 Deriving Constraints on Stable Systems

Constraints on the Traces of a Symbol

We briefly discuss how the goal axiom affects the traces of a symbol.

Referring to fig. 7.1, a symbol s, once created, has two classes of traces. If the creation is a goal, e.g. creating a file, the trace can end. Otherwise, it has to participate in other events as required by corollary (4.1). Being a symbol, the only kind of event that s can participate in is interacting with an agent. If events of this type are still not the goal, then the only other way for s to affect a goal is through an event initiated by the agent in the form

$$e(\omega)\ !\ r(\theta,\omega),$$

where ω is a physical or abstract object. For either case the simplest possibility is

$$\omega \in \text{realization}(\theta) \lor$$
$$\omega = \text{realization}(\theta).$$

In general, we expect θ, undergoing arbitrary transformation, becomes θ' and

$$\omega = f(\text{realization}(\theta')),$$

where f is arbitrary.

This shows both certain grounds for our intuition mentioned in the chapter on domain categories and its incorrectness for general cases.

Constraints on State Changes in Stable Systems

The assumptions about stable systems can be used in proving behaviors about them.

In particular, we observe that if a system is to be stable (in our restricted sense), its final state should be the same as its initial state. This includes two parts. If a relation holds at the start, it should hold at the end, again. If a relation is not known to hold at the start, it should not hold at the end, either.

Let the initial state for the i-th recurrence be $S_{initial,i}$ and the final state $S_{final,i}$. To prove the first part, we get the following formula from the goal assumption

$$\forall p\ [p \in S_{initial,i} \rightarrow (\exists e\ [e \in H(S) \land p \mapsto e\ ])],$$

for a recurrence e_i. Assuming e is the event enabled by p in H(S).

Assume

$$\neg p \in S_{final,i},$$

from

$$S_{final,i} = S_{initial,i+1},$$

we conclude

$$\neg p \in S_{initial,i+1}.$$

Considering formula (4.2) (also cf. the chapter on time), we have for event sequence $\|e_{i+1}\|$

$$e \in \|e_{i+1}\|.$$

But by definition of precondition,

$$\neg p \rightarrow \neg e.$$

Contradiction.[2]

[2]Note, e is an event class here. That is why it will be both in e_{i+1} and e_1

Chapter 8

Related Works in Formalizing Real World Domains

The formalization of real world domains is such **a** broad topic that to mention all of **the** related works, even at **a** very sketchy level, is virtually impossible. Moreover, in the **pre-**ceeding chapters, we have discussed in various places how this research is related with other works, especially when our basic concepts draw heavily upon the results of others. **The** following will be just a brief partial account. It is presented in about the same order as **the** chapters.

Abstraction Mechanisms in the Description Process

The three "golden braids" of abstractions: generalization, classification and aggregation, are the result of research in AI knowledge representation and programming languages **at** large. The idea of multiple levels and dimensions of abstraction is common knowledge today. Without the works regarding.each of these concrete forms of abstraction, there is no possibility of reflecting on a more general approach ([Bobrow and Collins], [Brachman], [Genesereth 1984], and [Greenspan]). Our notion of model mapping is a rationalization of the intuitions expressed by previous authors. It also clarifies some confusions. In particular, we made it clear that "omitting details" is important but not essential in an abstraction process. Any abstraction is done by selecting *relevant* facts and the relevancy often depends on causal or other lawful relations.

Our approach of model mapping, when being applied to determine primitive events, is close to Winograd's notion of "account" ([Winograd 1984]) where the recognition of the existence of an event at the level of the description is equated to selecting a series of "base events" on a totally ordered trace.

The formulation of our theory is influenced by the theory of abstract situations developed by Barwise and Perry ([Barwise and Perry]). Their theoretical mechanism for model building proved to be appropriate for our task. As a matter of fact, the observational model and abstractional model are very similar to their actual and factual situations.[1]

Basic Domain Categories

The ideas from metaphysics, particularly ontology, semantics, and language universals, are foundational in this kind of inquiry. AI knowledge representation research makes them more technical and consistent.

Works by Rosch et al. ([Rosch]) in cognitive psychology provide experimental evidence for the assumption of natural kind of objects. This indirectly supports hypothesizing the observational model and object categorizations.

Many authors argue for the existence of abstract objects and discuss their properties ([Yolton] [Wolterstoff] [Blau 1981a]). The distinction between different forms of states and changes gets constant attention from linguists ([Mourelatos]).

There are works that made serious effort in explicitly "dividing up the world" or categorizing things by inherent constraints. These include Schank's CD (conceptual dependency) ([Schank 1973] [Schank and Abelson] [Wilks]), Waltz's line drawing patterns, and others.

In Waltz's work, the domain categories and constraints are deduced, but its domain is too narrow and too amenable to mathematical (geometric, in fact) treatment to be taken as a model of even a considerable part of the real world.

Schank with his famous fourteen primitives ([Schank 1973]) has pioneered the analysis and categorization of events. Researchers holding the same tenet also worked out some analyses of objects ([Lehnert]). However, his theory is concerned with categorizing *event words in natural languages* in "semantic primitives" and therefore constitutes a claim about a cognitive process ([Winograd 1978]). Our framework, on the other hand, deals with artificial languages and serves at most as a rationally justified engineering tool.

Another difference between Schank and us is that his distinctions are prototypical rather than logical ones while we attempt an explicit formal domain model with constraints as logical consequences.

In conceptual dependency, the need for paraphrasing makes the primitives have many parameters ready to be filled in. These parameters reflect the rich contents of the primitives but the relations among primitives and parameters are left unexplained and the determi-

[1] According to [Barwise and Perry], actual situations exactly correspond to the real situations and factual situations "classify" real situations.

nation of the value of a parameter can be rather arbitrary. For example, for the primitive "mental transfer," MTRAN, it is not clear how it may be related to PTRAN (physical tranfer). ATRAN (abstract transfer) is just a mystery in terms of how it is related to observable changes (cf. our social relation). Furthermore, these primitives are simply used in programs and their exact formulations are never shown. As a matter a fact, it is hard to see how to infer the rate of health to be exactly somewhere between 10 and -10, even reasoning statistically. On the other hand, in our theory, it is those fundamental questions that are being asked - and the answers are axiomatized. For example, we showed the relation between physical forms and symbols (abstract forms) and relevant abstractions of events from one to another.

Serious discussions on the relation between a symbol and its reference date back to Frege. It is that basic idea and its various technical variations that encouraged our trichotomy of objects. For example the distinction made by [Smith] between $ and $ is exactly that made between our symbol and concept. However, his semantics is limited to the domain of mathematics and logic of computation, therefore, for him, a concept, as an abstract object, does not have a realization, which is the third layer in our scheme.

Maturana's works on autopoiesis ([Maturana and Varela]) cover many issues involving systems. In particular, he gives an illuminating account of the identities of composite objects, which we have discussed in the section on object identity.

Temporal Relation and Causation

Temporal relation is central in process description. Many schemes have been developed within and outside AI.

We did not adopt schemes like Petri-nets mainly because their non-linguistic nature makes them difficult to represent many complex relations, for example negation. The temporal logics by researchers such as Lamport and Owicki ([Lamport] and [Owicki]) are designed for proving certain properties of computation systems. They are rigorous but do not meet the versatile representation needs of describing real world processes.

Among AI researchers, Allen and McDermott both developed substantial frameworks for representing time and events. Allen's interval-based approach is adopted in our scheme. But [Allen] is more concerned with reasoning about temporal statements, using his sophisticated algorithm, than with devising a framework to accommodate ontological concepts other than events ("actions" in his terms). His scheme is very rich in expressing relations among time intervals but some very basic relations such as action class versus action instance are not discussed.

McDermott's temporal logic is more formal and concerns itself with many deep issues. But in his logic, temporal relations are formalized often without broader semantical and ontological perspectives. As a result, although he correctly rejected the situation calculus notion of events, the "running in circle" paradox remains. (One needs to be able to go up and down abstraction levels to explain it.) [Forbus] also points out that his framework is built on logic alone, and without certain ontological commitment, its reasoning capability cannot be powerful enough.

Both [McDermott] and [Allen] take terms as the logical form of events. This makes talking about relations between events easier but makes it harder to handle effects of events on objects involved. Our scheme tries to use both predicate formulas and terms as logical forms ([Davidson 1980]). This entails great complexity just in building the theory, but enables us to relate objects involved in different but relevant events. Our discussion on causation is such an example.

We have extensively reviewed approaches on causation both in and outside AI in the chapter on causation, and therefore will not repeat them here.

System Models

The original mathematical formulation of the system concepts is due to cybernetics. However, neither cybernetics nor, subsequently, general system theory use an information processing approach to analyze systems. As a result, they are unable to describe and analyze systems with complex behaviors, especially those with qualitatively identified states.

The research on operating systems in computer science touches on many issues pertaining to systems in general. For example, problems such as systems resource and systems goals were raised in that context. The difference between our work and theirs is that their emphasis is on algorithmic problem solution while we are more interested in a general discussion and axiomatization of the common properties and constraints of systems.

Attempts have been made by people to characterize systems in general from an information processing approach. [Holt 1971] analyzes a nursing school using a series of his system concepts such as roles and occurrences. The Delta project ([Holback-Hanssen et al.]) looked at numerous rules governing how components can influence one another in a system. However, in these works the ontological side of systems has been mostly neglected.[2]

Analysis of systems from both semantical and causal views is mainly done on physical systems. The credit goes to many AI researchers active in the problem of design

[2]Holt recently added physical objects to his conceptualization of system operations and is experimenting building large software systems based on this methodological tenet.

([De Kleer and Brown] [De Kleer 1984b] [Forbus] [Genesereth 1984] [Davis et al.]). Some of the analysis tries to identify the essential structure of systems, as we do. For example, [De Kleer and Brown] mentioned three categories of constituents of dynamic systems: components, conduits and materials. The first two correspond to our notion of components and the last non-components. Their intuitive account agrees with our formalization to a large extent.

It should be noted that our theory directly benefited from physics where many basic principles on systems are formulated in terms of differential equations of field theory. Careful readers will have noticed that our constraints on object existence and disappearance resemble the concept of divergence of field.

PART II

A Computational System For Constructing and Analyzing Specifications

Chapter 9

DAO as a Specification Language

9.1 An Overview of the Language DAO

Our language is an event-based system specification language. The term "event-based" emphasizes that in our domain model the concept of event will be in the central place. The language is named "DAO," which is the acronym for "Describing and Analyzing Operations." Our specification system (also named "DAO") integrates this language into a specification environment.

The syntax for DAO is written in a metasyntax so that it is easier to experiment with different language designs. A specification written in the DAO syntax is parsed by a recursive descent parser, which consists of a set of parsing procedures, implemented as Lisp functions. A parser generator can produce the set of parser functions from a version of DAO syntax. The parsing produces parse trees for pieces of specifications. The parsed specifications in the form of parse trees then can be type-checked and linked to one another. The parsing, type-checking and linking are called "compilation" in our system.

As for the notation, we note that we have adopted a *lisp-like* syntax for our language. A particularly salient feature of this notation is that it uses a lot of parentheses and puts a predicate in the same list as its arguments. This notation may seem strange to people who are accustomed to logic notations. But this notation is more practical since it is easier to be incorporated into an implementation based on a Lisp dialect.

9.1.1 General Characteristics of DAO

The basic tenet for the DAO language design is that the writing of specifications should be a disciplined activity [Dijkstra]. If a mechanical engineer has to be properly trained to draw engineering pictures so should a system engineer.

To achieve this, DAO stresses two things. First, it follows the general line in programming language design, i.e., making use of structuring tools, for example, building data type hierarchies such as generalizations, aggregation, and classification. Secondly, it imposes system concepts, for example stable system, goal, and information objects in specifications. These concepts form another layer of structuring. Moreover they entail checking power based on the analysis of the domain model.

To incorporate domain concepts into our language, DAO has many built-in constructs that are directly drawn from the domain model, e.g., agents, abstract object, and preconditions of events. These will be further analyzed in the semantics section. However, we note that there are two obstacles to accomodating our analysis into an artificial language. First, for any language, the number of constructs can not be too large. Otherwise people will simply refuse to learn it. Because of this, some fine distinctions can only be inferred by the DAO specification system, relying on heuristics.[1] We can not count on users to specify all of them. Secondly, at first sight some distinctions are not obvious or clear to common audience. These distinctions have been developed beyond immediately evident intuition. They are combined with a theoretical analysis of system behaviors, and the latter is something that has to be learned.

The Organization of Specifications

Logically, specifications are hierarchically organized into projects, groups and definitions. The implementation of groups are files. This makes group the basic unit of a coherent specification. For example, we may have a project for specifying businesses while having a group for restaurants.[2]

The definitions in a group are substantial descriptions. They may be used by other groups through an "inclusion" construct. In particular, some basic mathematical concepts and operations, some system concepts, and some standard spatial relations are stored in different groups for general use.

DAO is related in many ways to object-oriented programming languages and AI repre-

[1]In general, the use of heuristics in inferring the intended meaning of a piece of specification is undesirable in that we would like everythign made explicit in a formal specification. In fact, in the DAO system we mostly use domain constraints rather than heuristics to check out errors. Here we note that when we use heuristics, we are using them for an interactive checking system for detecting errors. The assumptions or guesses we make will always be known to the user because we will have to expalin why the system thinks a piece of specification has a problem. The user will have to examine these assumptions or guesses if he or she is to handle these errors. In this way, the use of heuristics will not entail any semantic processing unintended by the user.

[2]Like most of the ideas in syntax design, this is due to Winograd [Winograd 1984].

sentation languages. This will be shown in following sections. However, there is one thing DAO has intentionally left out, which is the control information. DAO uses all its expressive power for describing the "world per se" and leaves the control in processing the descriptions to a second level language. For example, the effect of "default" can be achieved in MRS (cf. Chapter 11) through a special language construct. In DAO, however, it has to be achieved by arranging assertions in the desired order because we do not have a construct to specify which axiom is to be taken as the default in system specifications.[3]

9.1.2 An Example Specification of Restaurant

We show an example of how our language is used. The following is part of a formal account of what happens in a restaurant. The emphasis is to show how conceptually distinct things are specified in categories and how logical assertions are connected by temporal connectives for representing complex system behaviors.

A more detailed account of the syntax and semantics of the language will appear later in this chapter. Therefore we will not describe the notation in detail here. In our example, we will be mainly relying on the reader's intuitions. The following notes on notation may provide the reader with a guide:

- The logic connectives with extended meanings (cf. the chapter on temporal relations): $\wedge$, $\vee$, and $\neg$, are represented as "and", "or", and "not."
- The temporal connectives defined in the chapter on time will be used here with the same meanings.
- A few self-explanatory key words are used in the example, including setOf, sequenceOf, setDifference, setUnion, const, var,... etc.
- The predicate "at" indicates a spatial relation. "In" means set membership or set inclusion.
- Comments are in (* ...), as in Interlisp.
- "(See <axiom>)" refers to an axiom <axiom> elsewhere to reduce duplication of assertions.
- Ellipsis dots ("...") indicate that parts of the specification are omitted for brevity.

[3]The assumption here is that when the knowledge base is incomplete, the inference engine will have to make some assumptions, which will depend on the process of inference. This is a potential source of inconsistency, of course.

Given this, the reader should be able to understand the general ideas in this piece of specification. Still, if he wants to follow its finer points, he may need to come back, after reading the succeeding sections.

The construct "history" is used for giving an overall picture of the system's behavior. The history of a restaurant says that waiters work in parallel, each waiter's service for customers is also in parallel, but customers come in and dine in sequence (which is denoted by "/" in our temporal notation).

```
[History restaurant:
 ((forAll w: waiter)
   (forSome cgSequence: SequenceOf customerGroup)
   ((forAll eg in cgSequence)(/)(dining eg w)))]
```

Next we define "customerGroup." A group of customers at a table is treated as a whole. It is assumed they do not change tables. Their current order reflects the dishes that are ordered but not served. The keyword "AgentClass" indicates the definition is for an agent class. We use similar keywords for information object class ("InfoObjClass"), concrete object class ("ConcObjClass") and so on.

```
[AgentClass (customerGroup: setOf person)
   (Const (table: a table; definingAxiom (at customerGroup table))
     (* occupying a table is more than physically being there. "At" is a simplification))
   (Var (currentOrder: setOf orderltem; definingAxiom (see dining)))]
```

Waiter and chef as agent classes are similarly defined.

Thirdly, order is defined as a set of order items which in turn are-names of dishes.[4] Names are a basic class of information objects of DAO. In terms of our object categorization they are symbols. (We do not want get into details of the physical form here so we will stay at this abstract level.) Every order item should be on the menu. Also, the name of a dish should be a dishName. Note that we use "dish.name" to mean the "name" property of a dish.

[4] Actually, an order item should be the combination of a dish name and the quantity. However, we make this simplification so that the presentation is easier to understand.

```
[InfoObjClass (order: setOf orderItem) ...]
[InfoObjClass (menu: SetOf dishName) ...]
[InfoObjClass (orderItem: dishName)
   (definingAxiom ((forAll orderItem: orderItem)
      (forSome menu: menu)(in orderItem menu)))]
[InfoObjClass (dishName: name)
   (definingAxiom ((forAll dish: dish)
      (forSome dishName:dishName)(dishName=dish.name)))]
```

"Dish" is a concrete object. Each dish has a name and price, which it shares with all dishes belonging to the same subclass. For example, steak is a subclass of dish. Clearly, by defining dish and order in our way, we won't make the mistake of "(order x (a dish));(cook chef (the dish))" (cf. the introduction).

```
[ConcObjClass (dish: primitive)
   (Const (subClassConst name: a name)
      (subClassConst price: a moneyAmount))]
```

We then define event classes. Primitive events are specified primarily in "NecessaryCondition" and "Consequence," which correspond to "precondition" and "consequence" in our operational domain model. The consequence of the event "order" is that an order is generated (indicated by "new" in the argument declaration) and received by the waiter.

```
(EventClass (order (a customerGroup)(a waiter)(new an order)) primitive
   (Consequence (at order waiter)
      (customerGroup.currentOrder = order)))
(EventClass (bring (x: individual)(y: individual)(an item)) primitive
   (NecessaryCondition (at item x))
   (Consequence (at item y)))
```

For the event of eating, we specify that its consequence is to make the food disappear. In choosing a basis for describing an operational system, we select relevant properties. In this case, we do not include the fact that eating makes the customer less hungry. The event of bringing is defined as a pure spatial change. The event "takeAway," removing plates containing the food, has the same type as the event class "eat" (the type is nullification, indicated by "nil"), which may be a problem. The problem is due to confusions generally existing between storage objects and their contents, e.g., a stack frame and its contents. As defined in this way, the types of primitive events can be easily inferred.

```
(EventClass (eat (a customerGroup)(nil a dish)) primitive
    (NecessaryCondition (at dish customerGroup)) ... )
(EventClass (takeAway (a waiter)(a location)(nil an item)) primitive
    (NecessaryCondition (at item waiter) ... )
```

"Cook" is a composite event. The chef cooks a dish according to order items. He first "looks at" the order item and then does the actual cooking which is an event "cookl"

```
(EventClass (cook chef (an orderltem)(new a dish))
    (definition ((sense Chef orderItem); (cookl Chef dish)! (dish.name = orderltem)))
where cookl is defined as
(EventClass (cookl chef (new dish: dish)) primitive
    (NecessaryCondition ((forsome kitchen)(inside chef kitchen)))
    (Consequence (at dish chef)))
```

Filially, we define the composed event "dining" which we call an "episode." To simplify the analysis, we declare all noncomponents and menus as "local," i.e., they will not interact with other objects. Note that the waiter brings dishes as many times as the number of courses, but the customer does not necessarily eat the dishes in the same order. The only constraint is, as said in the specification, a dish is first brought and then taken away.

```
(Episode (dining (eg: customerGroup)(w: waiter))
    (local all-non-component ... menu )
 (Definition ((...(bring w eg (a menu));
   (order eg w (an order));
   (bring w chef order);
    [((forAll orderltem in order)
          (&)(cook Chef orderItem (dish: dish))
        k
        [[(repeat for i: = 1 to numberOfCourses
              ([((bring w cg(dishes: setOf dish)) and (isSubset dishes dishesCooked) !
                 (cg.currentOrder <— (setDifference cg.currentOrder ((forAll d in dishes)(setUnion
d.name)];
              (takeAway w cg.table (dishes': (setOf dish] !
          (cg.currentOrder = {}) ]
        k
        ((forAll orderltem in order)(k)
          (eat eg (a dish suchThat (dish.name = orderltem];...)
      (Axioms [(forAll x: order)((order eg w x) < (bring w chef x ]
          (forAll dish)[(bring dish)< (takeAway dish)])))
```

9.2 The Syntax of the DAO Language

The basic structure of the syntax of DAO is, to some extent, a variation of that of Aleph, developed by Winograd [Winograd 1984]. Some explanations on the principles and ideas can be found in [Winograd 1984] and will not be presented here.

The full DAO syntax is in the appendix of this thesis. The syntax is designed for specifications in Lisp-like notations. This can be seen from the specification example above.

The syntax is designed for the use of a recursive descent parser with appropriate semantic actions for each parsed constructs. It is written in a metasyntax explained below.

We note that, here, we do not intend to give a full description of the language because it would be too long and would require a separate report. In this section and the section on the semantics of DAO we will concentrate on its novelties and difficulties. Standard answers to standard questions will not be spelled out here.

9.2.1 Metasyntax, Abstract Syntax, and Categories of Syntactic Classes

Because the syntactic notations are in constant flux, DAO uses a conventional set of meta symbols to ease the pain of rewriting grammars.

The meta syntax symbols used are:

1. X* zero or more occurrences of X

2. X-f one or more occurrences of X

3. X | Y | Z alternatives — one of them will be chosen

4. {X } X is optional

5. [...] is used for grouping, i.e., as parentheses.

6. <...>,<<...>> are parentheses for syntactic classes

7. @ is the prefix for an attribute of a syntactic class

8. =: and = connect LHS and RHS of a production.

Additionally, a string is a terminal symbol if it is *not* enclosed in angle brackets or prefixed with @.

Not all entries in a production rule are of equal importance in terms of semantics of the language. If an entry is semantically significant then there would be an attribute associated with it and some action will be taken at parsing time to let its instantiation in the specification be recorded as the value of that attribute. In our kind of parsing, only the values of these attributes, i.e., the information that is relevant to the semantic content of specifications, is left. These semantically significant entries form a syntax independent of the concrete notations for a language. They are so called *"abstract syntax."* The symbol "@" indicates an attribute, i.e., an entry in an abstract syntax.

The use of abstract syntax and meta syntax makes it possible to experiment with alternative syntactic notations. For that purpose, we implemented a parser generation package which proved to be very handy to use.

There are five categories of syntactic classes. They are categorized according to their semantic significance and processing characteristics.

In terms of implementations, each syntactic class corresponds to a LOOPS [Bobrow and Stefik] class. An instance of a syntactic class (a syntactic node in the parse tree) corresponds to a LOOPS object instance. An attribute corresponds to an LOOPS instance variable of an instance (cf. the chapter on internal representations). The attributes of a syntactic node will be attached to the instance variables of its corersponding LOOPS object.

The classes for substantial concepts, e.g., object class, collection types, etc. form the most common syntactic category. Their productions have the production symbol "=:". Only in this case, a syntactic node (as well as a LOOPS object instance) will be generated for a parsed instance of a syntactic class.

Another category is for choices in parsing. In this case, "=" instead of "=:" is used. The rules in this category are always single class names separated by "|".

The third category is the so called "pseudo classes." The rationale of the use of this category is based on implementation details and will not be explained here.[5]

The category of keywords need little explanation.They are enclosed by << ... >>. For ease of parsing, the *concrete syntax* uses many keywords at the beginning of each construct. As in most designs of concrete syntax, we put emphasis on the readability of the specifications. But we will not go into detailed examples here.

Another category which is also enclosed by << ... >> is associated with Lisp functions. Their production rules can not be found in the syntax but they are all straightforward. The reason they are not in the grammar form is purely one of efficiency considerations. For example, ATOM0 tests if a parser input is a non-NIL atom, FIXP tests if it is an integer, and ObjName tests if it is different from all the known keywords. (e.g., "a," "an," "if" and so on.)

As an example, we consider the a piece of a production

```
        <ObjClass> =: (<<ObjClassKEY>> @subtype)
       (<NamedDeclaration> @signature)
          <GENERALIZATIONS>....
```

The first item "(<<ObjClassKEY>> @subtype)" indicates that if an element in a list is recognized as a keyword for object classes (e.g., "InfoObjClass", "ConcObjClass", etc., the element will be the value of the attribute "subtype." The second entry is a non-terminal node. The second element in the piece of specification, which is necessarily a list, should be parsed into the syntactic class "NamedDeclaration" for the parse of the node "ObjClass" to succeed. The next element in the list will have to fit the production for the pseudo class "GENERALIZATIONS." However, there will not be a node generated for the element. Any attribute specified in the rule for "GENERALIZATIONS" will be attached to the node for the whole list, i.e., the node for the definition of the object class.

[5]For example, in the construct for object classes, <GENERALIZATIONS> is for pseudo classes. All pseudo class names use upper case letters–Interlisp distinguishes cases. The need for them comes from the parenthesizing in Lisp: The parser takes each list element one at a time recursively, but we do not want to create a separate node for each element at each parenthesizing level. In particular, we often want the attribute to be associated with the higher level node (in this case, the object class definition node) rather than put it at this level (in this case, it would be the "pseudo" class node if it were created). The technique of pseudo class provides a good solution to this.

9.3 Semantics of Language Constructs in DAO

According to the discussion in Part I, the semantics of a language amounts to translating a piece of notation into a logic expression *with a set of domain concepts as base terms.* For tractability, we would like the expression to be a first order logic expression. However, this is not always possible. Because, as far as processing is concerned, we can often write a subroutine that performs tasks involving higher order inference, it is not always necessary, either. In either case, a definition in formal semantics can be of help since it makes the meaning of the language constructs clear.

In the following, the actual translations will be omitted if they are obvious. A logic formula will be given only when there is some interesting point to discuss.

9.3.1 Definitions and Declarations

Here we are using our own terminology. The class "definitions" is very much like type declarations in ordinary programming languages. Object instance "declarations" are like variable declarations.

Object Definitions

The production rule for definitions of object classes looks like the following:

```
<ObjClass> =: (<<ObjClassKEY>> @subtype)
          (<NamedDeclaration> @signature)
          <GENERALIZATIONS> {(<Default> @default)}...
          <DEFINITION> <CONST> <VAR>
          <STATUS> <PROPERTY> <AXIOMS>.
```

The pseudo-syntactic classes are similar, except they use "=" instead of "=:" indicating that no actual nodes in the parse tree will be generated. For example, for the pseudo class <GENERALIZATIONS>:

```
<GENERALIZATIONS> =
          Generalizations (<<ObjName>> @generalizations)*
```

The subcategory of an object class, for example, order, is indicated by its subtype attribute, i.e., "@subtype" in the production, which can be "agent class," "information object class," "abstract object class" and so on. "Abstract object" exclusively refers to ψ_ψ, i.e., abstract content or concept. Information object can refer to both $\theta_\psi$and θ_ϕ, i.e., physical form or abstract form. This is because many events to θ_ϕ can be mapped to events to θ_ψ. By so doing we "save" one syntactic category in our language.

The signature attribute (@signature in the production) of an object class is of the form "named declaration," (as indicated by <NamedDeclaration> in the production) for example, "X: <type>". In this example, the first part, X, is the identifier of the class. <type> can be both simple or collection type. It is one of the generalizations of class X (or "superclasses of X" in the terminology of object-oriented languages). This particular generalization is placed in the signature because the class being defined is indexed mainly by this generalization.

For each class XX in the generalizations of X, we have

$\forall x\ X(x) \rightarrow XX(x)$.

If the optional attribute "@default" is filled, the object can be further categorized, e.g., for an information object, the user can specifically indicate whether he wants the object to be physical form or abstract form, and so on. (Otherwise, the system either has to figure out itself or it can not use any specific constraint.)

A "@definition" attribute, that is associated with the psudo class "DEFINITION", is usually a set of axioms specifying the "defining properties" of the class [Brachman]. Very few definition attributes actually give the necessary and sufficient conditions. The exceptions are the cases with the definition of composite events or relations.

For a better organization of information, all unary predicates and properties of a class are presented together with the class definition. Both predicates and properties are subcategorized according to their temporal features. If they do not change with time, they are called "constant" ("const" is the keyword) and "property.", corresponding to the pseudo classes "CONST" and "PROPRETY" respectively. Otherwise, they are "var" and "status" (correspondingly, the pseudo classes "VAR" and "STATUS"). For example, the color of an object, if considered not changeable, will be "const." If the color is red for that class of objects, then "red" can be a property.

Additional axioms about the class can be put in "axioms" attribute (and the corresponding syntactic class is "AXIOMS."

Most object types in DAO are declared in class definitions. The only dynamically declared type is "collection type." This means that an object instance (variable) can be declared to be of a collection type which is not previously defined. For example, if we have defined dish as a class then we can simply say (eat (A: person) (DishSet: setof dish)).

Collections are vital for a specification just as arrays are for ordinary languages. Recalling the discussion in Part I, collection objects have some nice properties. For example, some of its relations can be distributed to its elements.

Unlike arrays we do not usually refer to a particular member of a collective object explicitly. As a matter of fact, we do not even have the means to do so (except to give a definite description). However, collections can be dynamically formed and regrouped while their constituents remain in the system and retain their identities. This gives the descriptive framework much flexibility (and much complexity, too!).

Event Definitions

The definition of event classes can be introduced through three syntactic classes: <Simple-Event>, <CompositeEvent> and <Episode>. The production rules are in the appendix for DAO syntax. For all three of them, the signatures are lists of participants (or arguments). The type of each participant is specified in the form of a named type declaration, for instance, "(E x: X y: Y)" where the type of the first participant is X and that of the second is Y. The specific names, e.g., x and y here, are useful for later reference in the whole definition body.

Event category <SimpleEvent> corresponds to primitive events in our model. Its necessaryCond and consequence attributes correspond to precondition and consequence, respectively. When a consequence involves creation or nullification, the signature will be changed to reflect that consequence. For example, if we have

(E (new x: X))

then that would mean that there is an event class such that

$E(x) \Rightarrow [exist(x) \land X(x)]$.

"Composite events" and "episodes" both correspond to composite events in our model, but there is a difference. A composite event is simply a straightforward composition formed by temporal and logical connectives. As a result, all the arguments appearing in the definition body should appear in the signature. (cf. the definition for event cooking in the first section.) It is only a "shorthand" of the larger composition and it can be simply expanded into the composition. As a shorthand, the arguments of a composite event need not be the actual participants of the event. For example, it is perfectly legal to define a composite event (MarryFatherOfFatherOf (x: X) (y: Y)) as (Marry (x: X) (FatherOf (FatherOf ((y: Y)))).

Episodes, just looking at their logical forms, are not so much different from composition events. But they are used mainly to cover a more or less complete causal chain containing some system goals (cf. the chapter on system model). A signature of an episode mentions only the objects that initiate it or are affected by its final result. Many of the objects involved in between can be declared as local, so that their interactions with external objects can be ignored. Recalling our discussion of event abstractions, this is also like a primitive event. As a matter of fact, this is a particular abstraction mechanism with which DAO supports incremental specifications.

The definitions of function and relation classes are fairly straightforward. Both may be primitive or composite. The composite ones are to be expanded into primitive definitions.

Variable Declarations and Scoping Rules

Here we are using terminology from ordinary programming languages to mean our own concepts: "type" for class and "variable" for object instance.

Most types in DAO are declared in class definitions. The only dynamically declared types are collection types. The complexity related to the processing of this type will be discussed in the section on quantified events.

An object is always declared to be of a type or class. In other words, in DAO all objects are associated with a class "from birth." As a matter of fact, it can not change its type, too, which means the classification scheme is static. An object declaration may be accompanied with a "such that" clause. This is a further constraint in addition to the type. It is just an additional axiom.

Objects can be declared (or introduced[6]) in many places in a text, rather than only at the beginning (cf. dynamic allocation). In the restaurant example, the declarations for the dish and the dish sets all appear when the relevant events happen. We believe this notation is more natural.

However, the scoping rules are still static. This manifests itself in two ways. First, each declared variable has a scope which is determined on the basis of the innermost construct enclosing it. There are many constructs that have "bounding" power. For example, a variable declared in a repetition would take the repetition as its scope. Quantified formulas, "Let construct," each "definition" or "axiom" construct and, finally, each class definition also form a scope. The variable declared within the scope can be thought of as being declared exactly at the beginning of the scope. For example, in the restaurant specification, the object "DishesBrought" only makes sense inside the repetition. Secondly, all the types have to be defined in a group or included in the group for the group to use it. Othewise, the object of that type would be rejected at the compilation stage.

[6]We remind the reader that in a system specification, the declaration of an object does not create it. A declaration only brings the object to the attention of the specification writer or, in other words, "introduces" the object. Of course, the introduction of an object should conform to the general constraints on object creation and disappearance but that is a *separate* question. One may be talking about an object known to exist but with unknown identity.

9.3.2 Object and Predicate Reference

Object References

A straightforward way of referring to an object is by using its name or identifier. Similarly straightforward are references to standard mathematical values and elements of a sequence. Both are done as in ordinary programming languages. Referring to an element in a sequence is very much like referring to an element in an array.

Functional references for unary functions have an additional format which is like the format of Pascal records, i.e., "(a x)" can be written as "x.a". Functional references have a problem general to all languages having objects. Namely, the referents could be both mathematical entities (what we may call "pure values") or objects with identities (referring to objects in the computer or in the model of real world). The criterion of identity for pure values is just equality. Therefore, they can be computed directly. The non-value functional references of an object are dynamically defined by past events. To resolve this kind of reference will be a problem in specification simulation.

Quantified references[7] are convenient for representing certain types of sets. For example, (forAll x: X)(forAll y: Y)(f x y) would mean

$\{f(x,y) \mid X(x) \wedge Y(y) \wedge \text{defined } f(x,y)\}$,

where f is a function.

Event and Relation References

Instances of events and relations both have arguments. In the syntax, therefore, they are called argumented references. (cf. the appendix on DAO syntax.) They are not notationally distinguishable for two reasons. First, it would take an extra mechanism for doing so. Second and more importantly, it is because they are often treated the same in predicate compositions.

These are the alternative forms that "predicates" can take:

<Predicate> = <TemporallyConstrainedPredicate> |
  <TemporalPredicate> | <Predication> | <TemporalOrder> |
  <TemporalAssertion> | <Equality> | <ChangeFunVal> |
  <HistoryElement> | <ScopedPredicate> |
  <Repetition> | <WhileLoop> | <QuantifiedTemporalPred> |
  <QuantifiedPredicate> | <SeeConstruct> |
  <Equivalence> | <ConditionalPredicate> | <BinaryBoolPredicate>|

[7]Due to Terry Winograd [Winograd 1984].

<NegatedPredicate> | <BreakPoint>[8]

The predicates in DAO can be classified into three groups according to the connectives or operators they use.

The first group is traditional logic connectives, e.g., equivalence, conditionals, and formulas connected by boolean connectives. They do not need much explanation.

The second group is temporal ones. Each of them can have a direct counterpart in our temporal representation scheme. For example, "temporal assertions" are based on operators $\Diamond$, $\Box$ and $\#$, "temporal orders" are based on precedence operators, and temporally constrained predicates find the same key words in their formal definitions. Some syntactic classes have familiar programming language construct names. For example, "repetitions" and "while loops." This should not be surprising since we observe repetitions in the real world as well as in the running of programs.

The third group is quantified events. They will be discussed in greater length in the next subsection.

Finally, we mention the predicate class "function value change" (FunValChange). Intuitively, it amounts to an assignment statement in programming languages. But its role in an event-based description is much more limited. Formally, the meaning of an assertion such as

(f x) $\leftarrow$ val

can be defined as

let equal(f(x), u, t) for u and time t, for a certain event E(t),
equal(f(x), val, t+|E(t)|).

Note that an event always has to precede such an assertion.[9] It is assumed that the assertion is made against the duration of that particular E preceding it.

Quantified Predicates

There are three categories of quantifications in DAO. The quantified variables can be without any restriction, can be of some type (class), or, belong to a set. (cf. the production rule for <Quantification> in the syntax.)

There are standard translations for the latter two constructs. For the "class variable" case, we have, for instance, for

[8]The last item <BreakPoint> is a not a proper syntactic class in DAO. It can be inserted in the specification text for debugging purpose.

[9]The reader is reminded that in such a system,–real world system, any change has to be brought about by events.

(forAll x,y: X)(p x y),

a logic formula

Vx,y[X(x)AX(y)-p(x,y)]

and in the case of

(forSome x,y: X)(p x y),

a logic formula

3x,y[X(x)AX(y)Ap(x,y)].

Similarly, for "set variable" case,

(forAllx,y<EX)(pxy)

translates into

Vx,y[xEXA yeX ->p(x,y)l

and

(forSome x,yE X)(p x y),

into

3x,y[xeXA yeX) A p(x,y)]. ,<

However, quantified variables need not come from quantifications. There are implicitly quantified variables. For example, within the scope of a repetition, if a variable is declared, it will be effective for all repeating occurrences and will be a possibly new identity for each occurrence[10].

Formally, if we have a repetition

repeat e(x: £),

we will have (cf. the chapter on time)

3n: integer, 3X [II [xGXA i<n A e,-(x)]],

where X is a set of objects of class £. This is equivalent to

(3n: integer, 3X)(Vi)(3xGX)(;) [i<n A e;(x)].

There are implicit quantified variables in quantified formulas. For example, in

(forAll x in X) (e x (y: Y)), (9.2)

where e is an event, in most cases, the intuitive meaning would be that y will change each time a different x is selected. Therefore, it really amounts to

VxGX 3y [Y(y)A e(x,y)].

Allowing skolem functions, we have

3fVxEX[Y(f(x))Ao(x,f(x))].

But there is a danger in generalizing on this interpretation. We notice that formula (9.2) resembles simple formulas such as

[10] Of course, one can define it otherwise. But this interpretation is intuitively appealing.

$\forall x \in X\ e(A,x)$

or

$\forall x \in X\ e(A)$.

If we require a new relation or event for each new x, then we may not be able to keep the formula

$\forall x \in X\ e(A) \equiv e(A)$,

where A is an expression not containing x. But this is what the nice old first order logic allows us to have and we want to keep it.

Motivated by this and other reasons, we have two kinds of quantified predicates in our syntax. For the most part, one is for relations, and the other for events. They are notationally different, and the former is like ordinary logic notation. The definition of the latter is already in the chapter on time. The basic idea is that whenever we have

$$\forall x (\mathbf{op})\ E, \tag{9.3}$$

where **op** is an temporal operator, it is equivalent to having

$$\forall x\ \exists l\ (\mathbf{op})\ E(l), \tag{9.4}$$

where l is a variable ranging over spatiotemporal locations. In other words, there is very likely a *distinct* event for *each* x because in many cases the spatiotemporal constraints do not allow events to happen at the same time, the same place and with the same objects. For example, if we are to express the fact that the porter at the hotel entrance opens the door for every guest we should use (9.3). Only for a group of people coming in at the same time, is (9.2) appropriate.

Still, in (9.4) we did not exclude the possibility that for x_1 and x_2 in X, $e(x_1)$ and $e(x_2)$ could turn out to be the same event. For example, if what you want is to print a hardcopy for each file in a directory and you have two files with the same hardcopy appearance then it is hard to say what exactly we have asked the printer to do by such quantified events.

In DAO, as a design decision based on our experience, in order to express most common cases conveniently, and to process specifications easily, we adopt the convention that there will be a distinct event for each value of the universally quantified variable. In the existence of abstract objects, since it complicates the matter, we further require that in the case of

$\forall x \in X\ \exists y \in Y[e(x,y)]$

(the existential variable y and its set Y may be implicit), X and Y should form a one-to-one mapping. Note that in the case of object creation and nullification, this is not the result of our design but has to be the case.

Lastly, we mention an interesting phenomenon in notations for quantification, which shows how they are designed to achieve maximum conciseness and expressiveness. Ordinarily, if one has a quantification

(forAll x in X)(forSome y...)(exy),

then y would be either restricted in type (class) or be restricted to a known set. The idea is that y already exists when the event starts. It sounds senseless to say something like (forAll x in X)(forSome y in (Y: setof YY)).

On the hand, when we have creation events in a quantified event and we would like to refer to the created collection (since it is created through a set of events) we can not get a handle on it. For example, the specification of a restaurant given in a previous section uses two quantified events for the cooking of dishes and eating of them. Thinking carefully, you will find that what the customer eats need not be the dishes cooked *according to his order* because it may so happen he comes in first and has ordered a steak, the chef has cooked the steak for him, but the steak is given to another customer. The quantifications used in both formulas make it impossible to establish the identities of objects outside the scope of quantification, although this confusion might be exactly what some specification writers want for a restaurant. Normally, we would feel that the identity of the dishes that are cooked and those that are eaten should be the same. Efforts spent on finding a plausible notation have met many failures, until we found that the above seemingly senseless expression is grammatical and that it can serve our purpose! This turns out to be very useful and it is used in our actual specification for the restaurant example.

Object Hierarchy and Property Inheritance

As in many object-oriented programming languages, the object classes form a lattice in DAO. The property inheritance is logical implication rather than overwriting. That is, if there is a super class XX of class X and a property p then

Vx[XX(xHp(x)] -Vx[X(x)->p(x)].

Since

Vx[X(x)->XX(x)]

by definition. The above seems to be a tautology but we note that there are languages where this feature can be violated.

There can be overloading for event and relation names. That is, a predicate reference (p x y) can resolve to different definitions according to the classes of x and y (i.e., the classes with which they are created). For both properties and events multiple inheritance is allowed as long as there is no any conflict. Otherwise, it is an error that the user has to correct. For example, in the case of (p x y), we may have defined (p (u: XX)(v: Y)) and (p (u: X)(v: YY)). If we find out at run time that the class of x is X and the class of y is

Y then we have a conflict. If we did not define one of them or if we have also defined (p (x: X)(y: Y)) then we will be able to resolve it.

Since we can not determine the classes for all object references at compile time, a dynamic resolution scheme is used.

Chapter 10

DAO as a Specification System

10.1 Overview of the Integrated Specification System

DAO is an integrated specification system. It starts from facilities for users to create and modify specifications and carries out various checking procedures on the resulting specifications.

The architectuer of the DAO system is shown on Fig. 10.1. The functions of each "box" in the figure will be explained in detail in the remainder of the thesis. Here we briefly comment on their interrelations and the characteristics of the whole system. In the following we visit each "box" in turn. Note that the sizes of the actual system components are not proportional to the sizes of the boxes.

The first system component is responsible for producing a piece of specification text. Its design is geared to the kind of disciplined specification style we advocate. E.g., menu items are in the same hierarchy as language categories. This component is built on top of two Interlisp-D editors [1], but it has features that go beyond managing a particular version of text. For example, there is a hierarchical organization of specifications, in terms of projects, groups and definitions, managed by this component. The component keeps older versions for backup. It also allows various ways of viewing the specifications.

Owing to the LISP dynamic environment, this component can be reentered at any time during the development of the specification, as easily seen from the backward arrows on the figure. A particular piece of undesired specification can be modified, and put through all the necessary processing stages to get to where it is needed. Thus the system is incremental.

The product of the first stage is a readable and well organized document. Its structure is imposed by the editing environment which incorporates the DAO language. But it can

[1] It is interfaced to both TTYINEDIT and DEDIT.

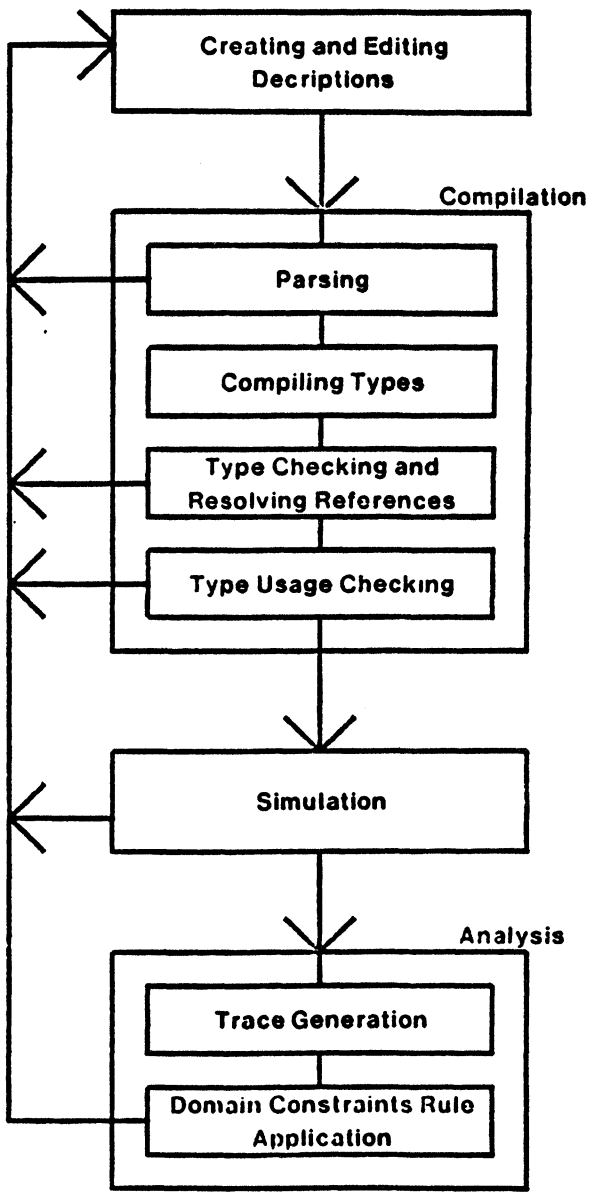

Creating and Editing Decriptions
Compilation
Parsing
Compiling Types
Type Checking and Resolving References
Type Usage Checking
Simulation
Analysis
Trace Generation
Domain Constraints Rule Application

Figure 10.1: The Architecture of DAO System

not be processed by a mechanical checking routine. The compilation stage parses the specification texts and, if successful, puts them in the form of parse trees. The content of the specification is now treated as type (class) definitions, variable (object) declarations, and references (of events, relations and objects). All the references are linked to their definitions. All linked parse trees in a specification form a hierarchy, with the system history at the top and the various primitive definitions at the bottom. This will be called the *"history tree"*. If there is an error detected, appropriate messages will be displayed. Facilities are provided to help identify the nature of the problem.

The last inner box in the compilation features a checking method not usual for ordinary languages. This is due to the special nature of real world system specifications. Real world system specifications are often concept-intensive but algorithm-meager. Type checking can find out if a used type is defined. But, finding out that a defined type is not used is also informative. It may indicate inappropriate abstraction levels or incompleteness of specifications.

Once in the form of a history tree, the specification can be further processed. In particular, this means simulated. Simulation is a very complex process involving complex mechanisms, for example, a special purpose theorem prover. Here it suffices to point out two of its major functions. First, it detects impossibilities and inconsistencies in a specification by actually going through the specification "step by step." Second, it produces a set of "traces" or possible sequences of events.

This set of traces is the proper input to the last stage, analysis. Here, checking rules are applied to individual objects in individual traces to discover violations of domain constraints. A specification that passes these stages of validation is not completely clear of bugs, but many problems will have been discovered and fixed.

Characteristics of the DAO System

The DAO system is built along the following guidelines, which become salient features of the system.

1. It is incremental. Changes can be made locally, and at almost any time. If, in simulating a specification, the user finds that a definition need be changed, he can edit the definition, parse it, resolve references in it, and the simulation will then use the new definition.[2]

[2]Of course, this may require that other parts of the specifications are recompiled, too. The system keeps information concerning all the places where a definition is used. But, of course, not all the places need be modified.

2. It is interactive. The user can inquire about the system state at many places. The system can also ask the user questions or ask for directives. Because the checking criteria are often heuristic in nature, this is often quite necessary.

3. It is display oriented. At each step, the major data structures have one or more graphic forms. This makes the semantic processing easier to follow.

4. It is validation oriented. In a sense, we do not care about how the specified system would be realized. The one and only concern of this system is to "get the story right." At every stage, some checking is done. An error is reported and can be fixed as soon as possible. The final result is not a running system but a somewhat better, more trustworthy specification.

5. It is debugging oriented. The system is not only interested in finding errors. It also provides various capabilities handling errors. For example, we can not only break parser functions in parsing specifications, but also the inference process and the simulation process can all be broken for particular rules or particular definitions. Once in a break, the system state can be displayed in a concise form for a quick inspection, or in the exact internal representation for locating more obscure problems.

10.2 A Scenario

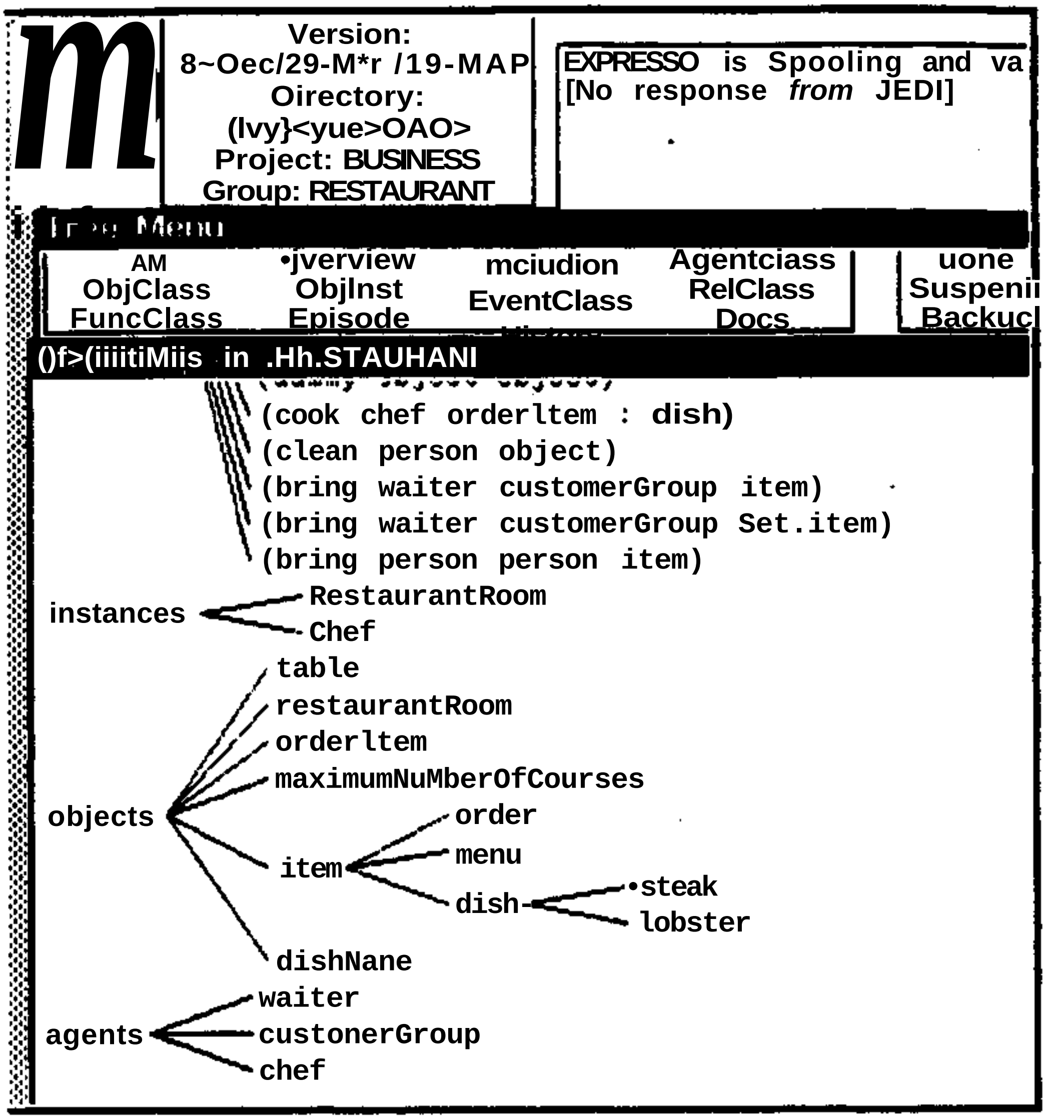
Before we systematically introduce the reader to the individual components of the integrated specification system DAO, we describe a detailed scenario of writing and checking a specification in DAO.

This is from a terminal session. The user interacts with the system through keyboard type-ins and mouse buttons.

We assume that the user has already loaded in the whole system including the specification he wrote earlier. Now he is looking at a screen, part of which you can see in the Figure 10.2.

At the upper left corner, there is the logo for the system DAO. Clicking that part, a top level menu will pop up. To its right, the small window displays the current project "BUSINESS," and the current group "RESTAURANT" that one can work on. [3] Clicking an item in this window, a menu will pop up, too. For example, the menu for "project" will let one create a new group, delete an old group, display all the groups in the current

[3]For exact definitions of terms such as project and group, you have to refer to the chapter on DAO syntax. But it should be self-evident what they mean in this context.



EXPRESSO is Spooling and va
[No response *from* JEDI]

```
(cook chef orderltem : dish)
(clean person object)
(bring waiter customerGroup item)
(bring waiter customerGroup Set.item)
(bring person person item)
instances    RestaurantRoom
             Chef
             table
             restaurantRoom
             orderltem
             maximumNuMberOfCourses
objects                        order
             item              menu
                               dish        •steak
                                           lobster
             dishNane
             waiter
agents       custonerGroup
             chef
```

Figure 10.2: DAO Logo and Definition Browser

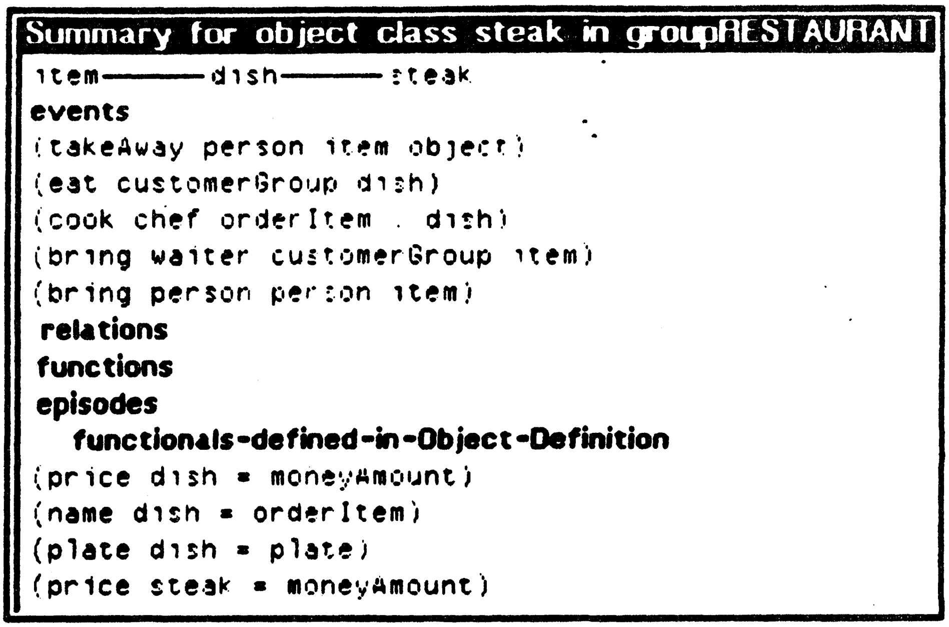

```
Summary for object class steak in groupRESTAURANT
item———dish———steak
events
(takeAway person item object)
(eat customerGroup dish)
(cook chef orderItem . dish)
(bring waiter customerGroup item)
(bring person person item)
relations
functions
episodes
  functionals-defined-in-Object-Definition
(price dish = moneyAmount)
(name dish = orderItem)
(plate dish = plate)
(price steak = moneyAmount)
```

Figure 10.3: Summary of Definitions Involving "steak"

project, etc. The menu for Group is much larger. We can show the definitions in a group, parse them, type check them, simulate them, and so on.

The classes in the specification of a restaurant are shown in the window below. They are grouped into categories such as objects, events, agents, etc. You may notice that the objects are displayed in the form of a tree. This shows the hierarchical relationship between the classes of objects.

The display of defined classes can take other forms. Figure 10.3 shows a "summary" for the class steak. What you see is that the class steak and the generalization classes of the class steak displayed at the top and all the relations, events, and functions involving an object of the class steak are shown below. Don't be surprised if you do not see "steak" in the event signature "(takeAway person item object)." Note that since "steak" is a subclass of "item," it can be wherever the class item can be.

You can edit a definition by clicking the name of that definition in display. If you would like to create a new definition, click an item in the "tree menu window." This will give you a template definition of the desired type for you to work on.

Once you are done with editing, you can parse the definition. If the parse is successful, you can view the parse tree in graph form or list form as shown in Figure 10.4 and Figure 10.5. The parsed structure may not be what you would like to have. So you can browse the parse tree by expanding a non-terminal node or simply display that node (with its children)

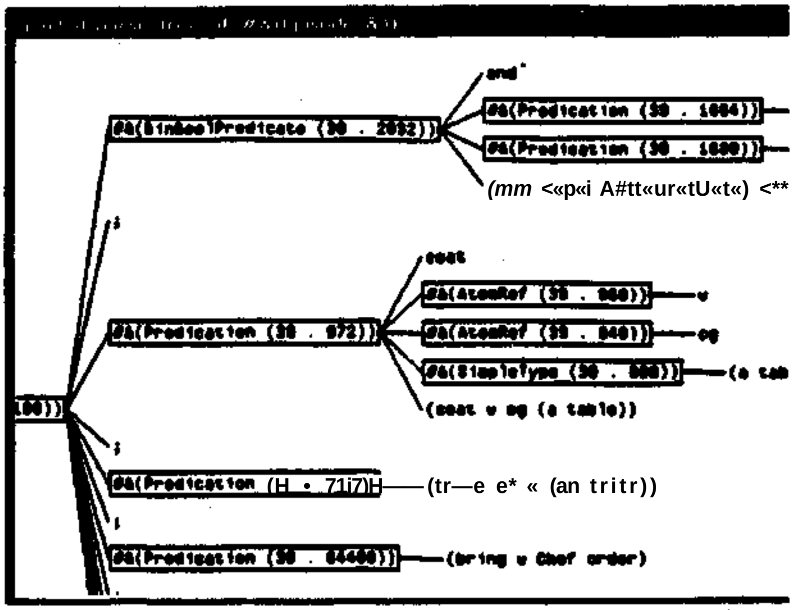

and
#&(BinaryPredicate (38 . 2032))
#&(Predication (38 . 1084))
#&(Predication (38 . 1020))
;
seat
#&(AtomRef (38 . 960))
w
#&(AtomRef (38 . 940))
cg
#&(SimpleType (38 . 980))
(a tab
(seat w cg (a table))
#&(Predication (38 . 972))
;
#&(Predication
;
#&(Predication (38 . 84400))
(bring w Chef order)

Figure 10.4: Graph Form of a Parse **Tree**

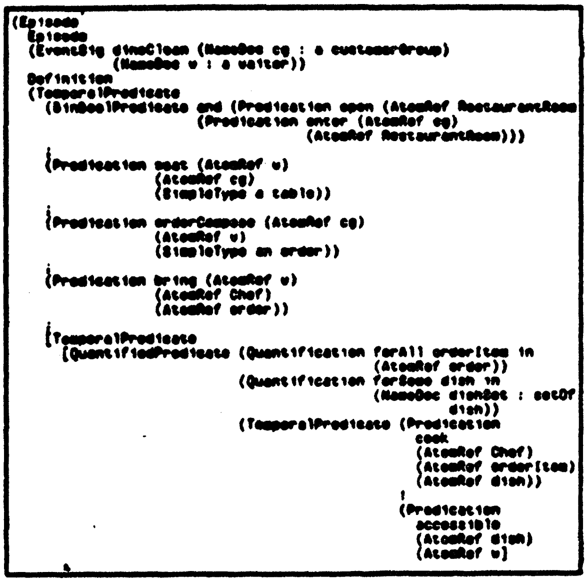
or examine its related grammar rules.

If the parse fails, you can view the partial parse tree and have a rough idea **of where** things go wrong. The next thing to do is to invoke the user break window (Figure 10.6). It automatically knows what type of definitions you sure trying to parse and displays relevant parser functions for you to break. You then parse the definition again. When it breaks, you can parse the suspected part of the definition and edit that part until the parse is successful.

The next thing to do is to type check and resolve references. Figure 10.7 shows a result of this. If you compare it with figure 10.5, you will find what was (AtomRef w) is now (NameDec w: a waiter). This is because the node for (AtomRef w) has been linked to what it refers to and the display procedure follows the link to find the referred object.

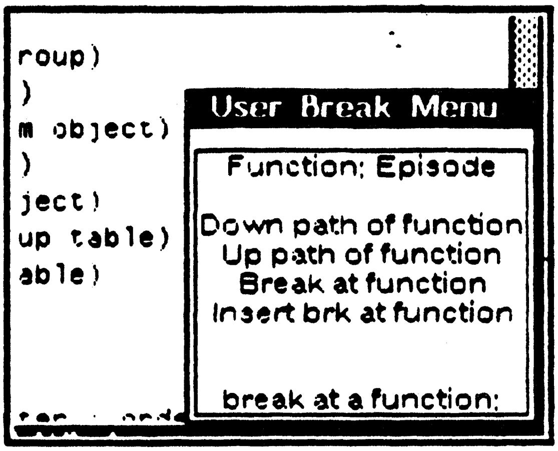
We now can start simulating the specification. The first thing DAO would do is display a flow graph of the events and relations specified, which will be called "history graph." This is in Figure 10.8. Note that this is not a tree. The arcs with arrows mean sequentially connected events or relations. The symbol "&" means concurrently, as has been introduced in the chapters on temporal representation and the DAO language.

The simulation will show what it is doing as it goes along. A sample of that running trace is in Figure 10.9. Potential errors are in bold face. The display has been made very concise. E.g., a universally quantified variable ranging over set S will be displayed as $x@S. The simulation is done by proving the preconditions of events and asserting the consequences of



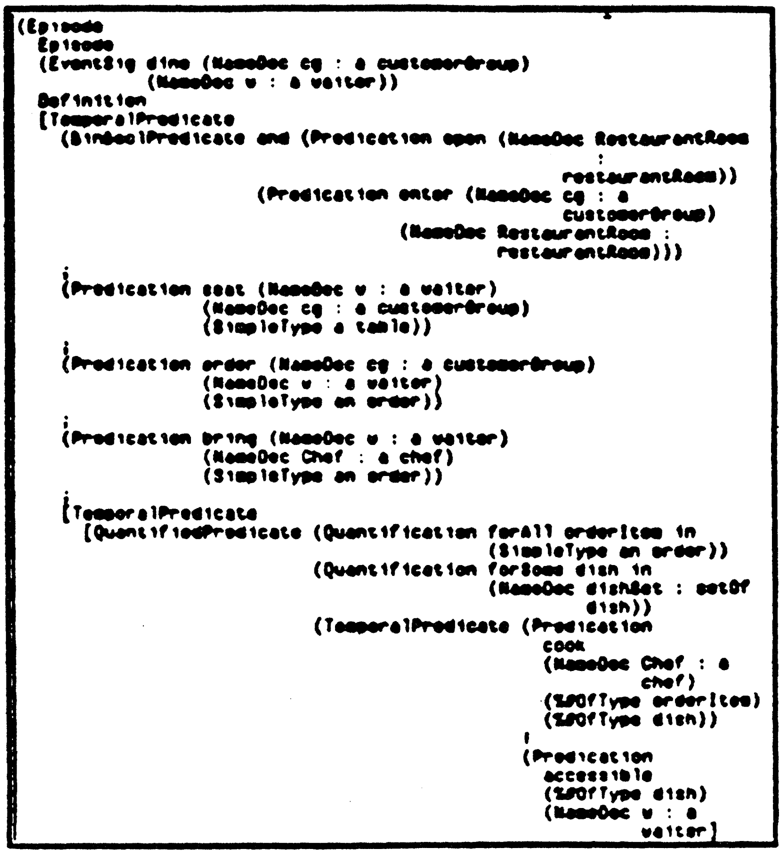
```
(Episode
  Episode
  (EventSig dineClean (NameDec cg : a customerGroup)
             (NameDec w : a waiter))
  Definition
  (TemporalPredicate
    (BinBoolPredicate and (Predication open (AtomRef RestaurantRoom
                          (Predication enter (AtomRef cg)
                                             (AtomRef RestaurantRoom)))

    ;(Predication seat (AtomRef w)
                      (AtomRef cg)
                      (SimpleType a table))

    ;(Predication orderCompose (AtomRef cg)
                      (AtomRef w)
                      (SimpleType an order))

    ;(Predication bring (AtomRef w)
                      (AtomRef Chef)
                      (AtomRef order))

    ;(TemporalPredicate
      [QuantifiedPredicate (Quantification forAll orderItem in
                                           (AtomRef order))
                           (Quantification forSome dish in
                                           (NameDec dishSet : setOf
                                                    dish))
                           (TemporalPredicate (Predication
                                                cook
                                                (AtomRef Chef)
                                                (AtomRef orderItem)
                                                (AtomRef dish))
                                              ;
                                              (Predication
                                                accessible
                                                (AtomRef dish)
                                                (AtomRef w]
```

Figure 10.5: List Form of a Parse Tree



```
roup)
)
m object)
)
ject)
up table)
able)
```

Figure 10.6: User Break Window



```
(Episode
  Episode
  (EventSig dine (NameDec cg : a customerGroup)
             (NameDec w : a waiter))
  Definition
  [TemporalPredicate
    (BinBoolPredicate and (Predication open (NameDec RestaurantRoom
                                                                  :
                                                          restaurantRoom))
                          (Predication enter (NameDec cg : a
                                                          customerGroup)
                                   (NameDec RestaurantRoom :
                                                 restaurantRoom)))
     ;
    (Predication seat (NameDec w : a waiter)
                 (NameDec cg : a customerGroup)
                 (SimpleType a table))
     ;
    (Predication order (NameDec cg : a customerGroup)
                 (NameDec w : a waiter)
                 (SimpleType an order))
     ;
    (Predication bring (NameDec w : a waiter)
                 (NameDec Chef : a chef)
                 (SimpleType an order))
     ;
    [TemporalPredicate
      [QuantifiedPredicate (Quantification forAll orderItem in
                                            (SimpleType an order))
                           (Quantification forSome dish in
                                            (NameDec dishSet : setOf
                                                   dish))
                           (TemporalPredicate (Predication
                                                 cook
                                                 (NameDec Chef : a
                                                          chef)
                                                 (%OfType orderItem)
                                                 (%OfType dish))
                                               ;
                                               (Predication
                                                 accessible
                                                 (%OfType dish)
                                                 (NameDec w : a
                                                          waiter]
```

Figure 10.7: Resolved Parse Tree

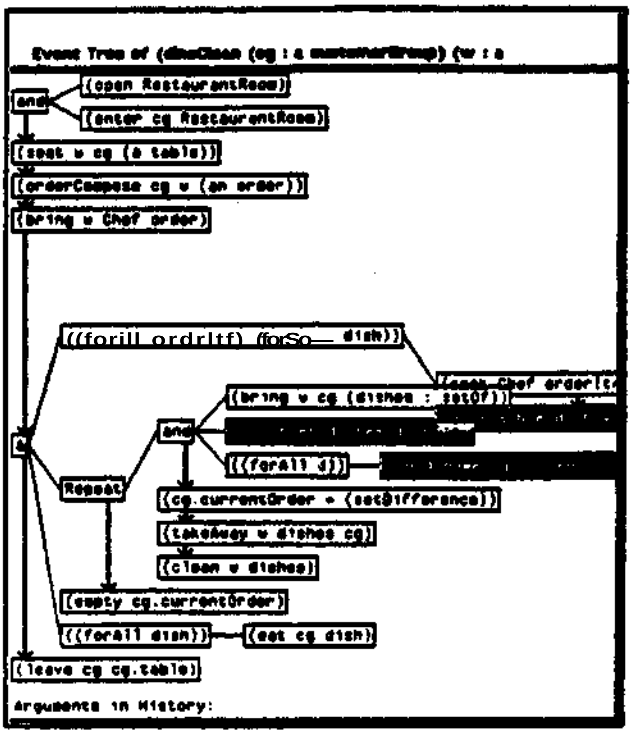


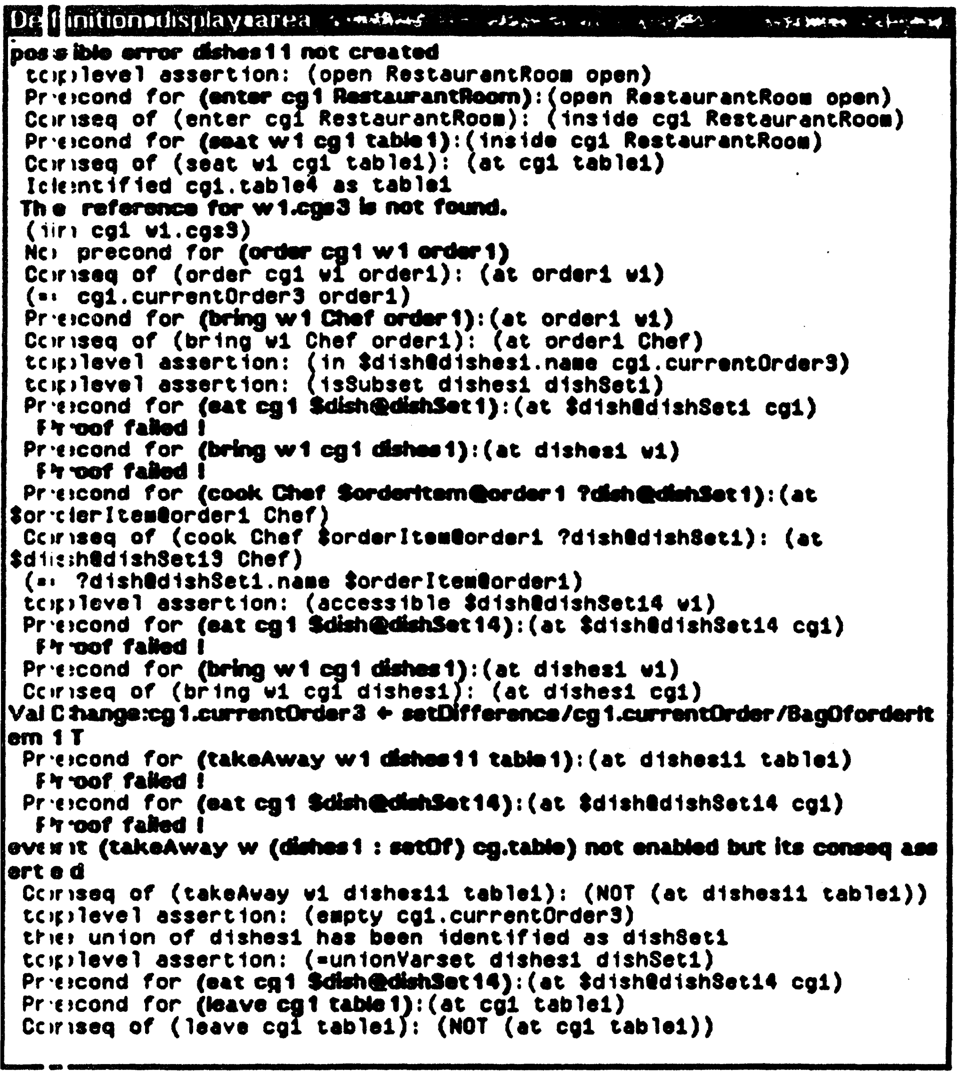
Figure 10.8: The Chart for Simulation Display

proved events as can be seen from the "precond" (precondition) and "conseq" in the figure. The proving process can be traced, too. The process of simulation, along with the same example will be explained in detail in the chapter on simulation.

If the inference does not give you what you wanted, you may want to examine the rules and facts in the knowledge base. The rules are in a global knowledge base, while the facts arc associated with the context where they are true. If you look at the menu in Figure 10.10, there are items for showing mles and for showing contexts. You get to the context first and then you can examine facts.

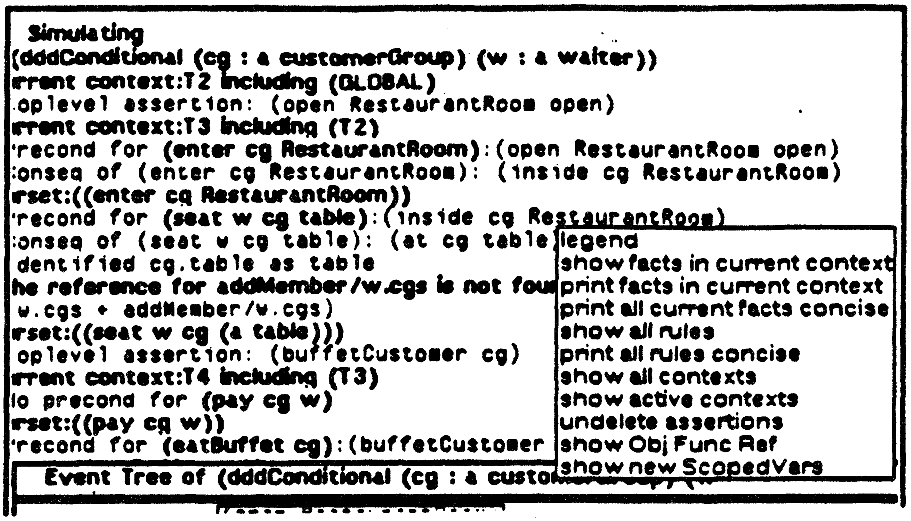
Some rules and facts are displayed in Figure 10.11. This is too difficult to read. However, sometimes they are useful, so we keep them. You can get a much nicer looking display, as in figure 12. Since we only store one version in the knowledge base, this cleaner view is converted by the system from the somewhat messier one.

For each event or relation to be simulated or displayed in the flowchart, one can do several things, as indicated in the menu shown in Figure 10.12. One can open a window for the definition to which this assertion refers. Once you get to the definition, you can ask for proving a particular relation when you get into a preset break. This is a convenient feature for debugging.



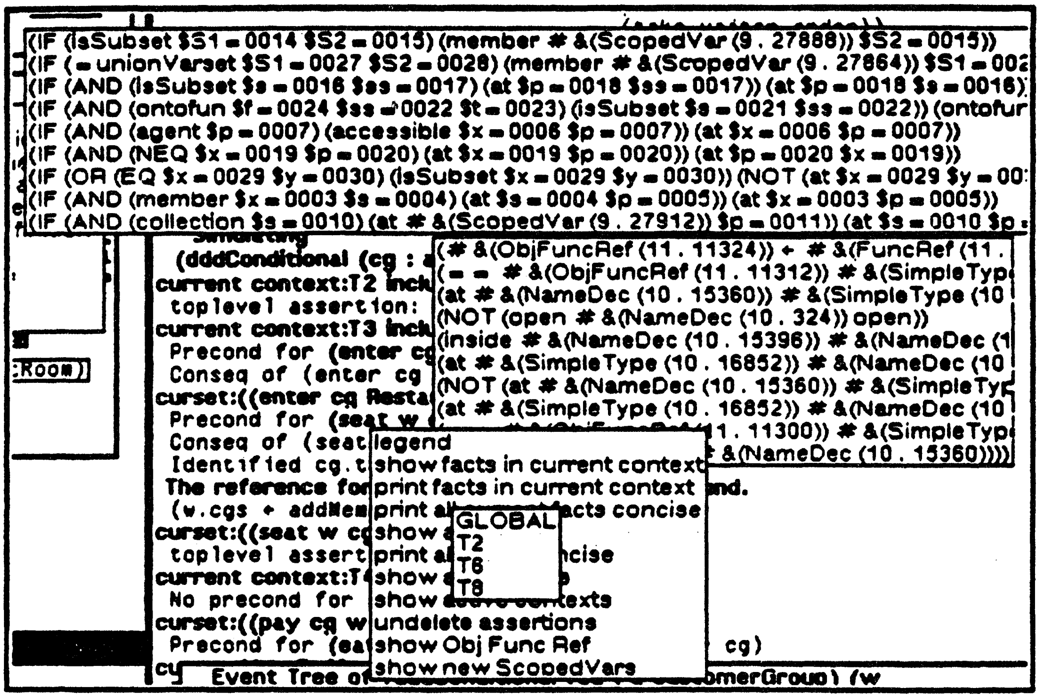
```
possible error dishes11 not created
 top level assertion: (open RestaurantRoom open)
 Precond for (enter cg1 RestaurantRoom):(open RestaurantRoom open)
 Conseq of (enter cg1 RestaurantRoom): (inside cg1 RestaurantRoom)
 Precond for (seat w1 cg1 table1):(inside cg1 RestaurantRoom)
 Conseq of (seat w1 cg1 table1): (at cg1 table1)
 Identified cg1.table4 as table1
 The reference for w1.cgs3 is not found.
 (in cg1 w1.cgs3)
 No precond for (order cg1 w1 order1)
 Conseq of (order cg1 w1 order1): (at order1 w1)
 (= cg1.currentOrder3 order1)
 Precond for (bring w1 Chef order1):(at order1 w1)
 Conseq of (bring w1 Chef order1): (at order1 Chef)
 top level assertion: (in $dish@dishes1.name cg1.currentOrder3)
 top level assertion: (isSubset dishes1 dishSet1)
 Precond for (eat cg1 $dish@dishSet1):(at $dish@dishSet1 cg1)
  Proof failed !
 Precond for (bring w1 cg1 dishes1):(at dishes1 w1)
  Proof failed !
 Precond for (cook Chef $orderItem@order1 ?dish@dishSet1):(at
$orderItem@order1 Chef)
 Conseq of (cook Chef $orderItem@order1 ?dish@dishSet1): (at
$dish@dishSet13 Chef)
 (= ?dish@dishSet1.name $orderItem@order1)
 top level assertion: (accessible $dish@dishSet14 w1)
 Precond for (eat cg1 $dish@dishSet14):(at $dish@dishSet14 cg1)
  Proof failed !
 Precond for (bring w1 cg1 dishes1):(at dishes1 w1)
 Conseq of (bring w1 cg1 dishes1): (at dishes1 cg1)
Val Change:cg1.currentOrder3 ← setDifference/cg1.currentOrder/BagOforderIt
em 1 T
 Precond for (takeAway w1 dishes11 table1):(at dishes11 table1)
  Proof failed !
 Precond for (eat cg1 $dish@dishSet14):(at $dish@dishSet14 cg1)
  Proof failed !
event (takeAway w (dishes1 : setOf) cg.table) not enabled but its conseq ass
erted
 Conseq of (takeAway w1 dishes11 table1): (NOT (at dishes11 table1))
 top level assertion: (empty cg1.currentOrder3)
 the union of dishes1 has been identified as dishSet1
 top level assertion: (=unionVarset dishes1 dishSet1)
 Precond for (eat cg1 $dish@dishSet14):(at $dish@dishSet14 cg1)
 Precond for (leave cg1 table1):(at cg1 table1)
 Conseq of (leave cg1 table1): (NOT (at cg1 table1))
```

Figure 10.9: The Display of Simulation Steps



```
 Simulating
(dddConditional (cg : a customerGroup) (w : a waiter))
rrent context:T2 including (GLOBAL)
.oplevel assertion: (open RestaurantRoom open)
rrent context:T3 including (T2)
'recond for (enter cg RestaurantRoom):(open RestaurantRoom open)
:onseq of (enter cg RestaurantRoom): (inside cg RestaurantRoom)
rset:((enter cg RestaurantRoom))
'recond for (seat w cg table):(inside cg RestaurantRoom)
:onseq of (seat w cg table): (at cg table
 dentified cg.table as table
he reference for addMember/w.cgs is not fou
 w.cgs + addMember/w.cgs)
rset:((seat w cg (a table)))
 oplevel assertion: (buffetCustomer cg)
rrent context:T4 including (T3)
lo precond for (pay cg w)
rset:((pay cg w))
'recond for (eatBuffet cg):(buffetCustomer
  Event Tree of (dddConditional (cg : a custo
```

Figure 10.10: Menu for Simulation

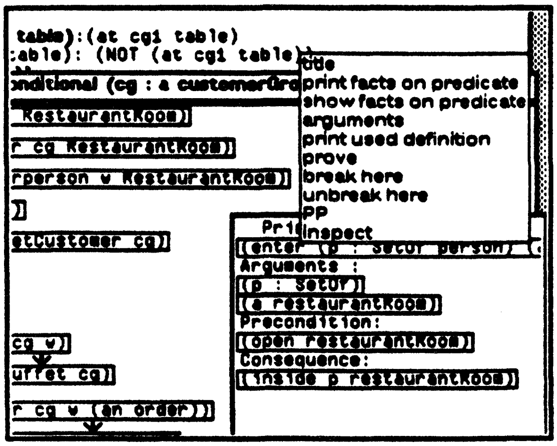


```
(IF (isSubset $S1 = 0014 $S2 = 0015) (member # &(ScopedVar (9 . 27888)) $S2 = 0015))
(IF (= unionVarset $S1 = 0027 $S2 = 0028) (member # &(ScopedVar (9 . 27864)) $S1 = 002
(IF (AND (isSubset $s = 0016 $ss = 0017) (at $p = 0018 $ss = 0017)) (at $p = 0018 $s = 0016)
(IF (AND (ontofun $f = 0024 $ss = 0022 $t = 0023) (isSubset $s = 0021 $ss = 0022)) (ontofur
(IF (AND (agent $p = 0007) (accessible $x = 0006 $p = 0007)) (at $x = 0006 $p = 0007))
(IF (AND (NEQ $x = 0019 $p = 0020) (at $x = 0019 $p = 0020)) (at $p = 0020 $x = 0019))
(IF (OR (EQ $x = 0029 $y = 0030) (isSubset $x = 0029 $y = 0030)) (NOT (at $x = 0029 $y = 00
(IF (AND (member $x = 0003 $s = 0004) (at $s = 0004 $p = 0005)) (at $x = 0003 $p = 0005))
(IF (AND (collection $s = 0010) (at # &(ScopedVar (9 . 27912)) $p = 0011)) (at $s = 0010 $p
```

```
(# &(ObjFuncRef (11 . 11324)) + # &(FuncRef (11 .
(= = # &(ObjFuncRef (11 . 11312)) # &(SimpleTyp
(at # &(NameDec (10 . 15360)) # &(SimpleType (10
(NOT (open # &(NameDec (10 . 324)) open))
(inside # &(NameDec (10 . 15396)) # &(NameDec (1
(at # &(SimpleType (10 . 16852)) # &(NameDec (10
(NOT (at # &(NameDec (10 . 15360)) # &(SimpleTyp
(at # &(SimpleType (10 . 16852)) # &(NameDec (10
1 . 11300)) # &(SimpleTyp
&(NameDec (10 . 15360))))
```

```
  (dddConditional (cg : a
current context:T2 inclu
  toplevel assertion:
current context:T3 inclu
  Precond for (enter cg
  Conseq of (enter cg
curset:((enter cg Resta
  Precond for (seat w
  Conseq of (seat
  Identified cg.t
 The reference for                              nd.
  (w.cgs + addMem
curset:((seat w cg
  toplevel assert
current context:T4
  No precond for
curset:((pay cg w
  Precond for (ea                               cg)
cy  Event Tree of                               merGroup) (w
```

Figure 10.11: The Rules and Facts in Internal Representation



```
table):(at cg1 table)
:able): (NOT (at cg1 table))
onditional (cg : a customerGro
RestaurantRoom)]
r cg RestaurantRoom)]
rperson w RestaurantRoom)]
)]
etCustomer cg)]
cg w)]
uffet cg)]
r cg w (an order))]
```

```
Pri
[(enter (p : setOf person) (
Arguments :
[(p : setOf)]
[(a restaurantRoom)]
Precondition:
[(open restaurantRoom)]
Consequence:
[(inside p restaurantRoom)]
```

Figure 10.12: Display of Used Definition

The system, upon successful simulation, will generate traces (in DAO's system model sense). One typical trace is in the upper half of Figure 10.13. If you compare this trace with the running trace of simulation, you may find that objects that were of the same identity but under different names, are now represented by the same name. This is necessary for the analysis.

The lower half of figure 10.13 shows the result of analysis. Each of the last three lines reports a not nullified object. For "restaurantRoom," the error is because of a wrong declaration. It was not declared as a component. For "cg," it is a real problem: Somehow our simulation shows that the event of his leaving the restaurant can not be enabled. The object order appears here, too, because it is incorrectly treated as a physical form of an information object. We note that symbols need not be explicitly nullified, only physical objects do.

10.3 The DAO Specification Environment

The DAO specification environment is an interactive computing environment that enables a user to write, read and analyze specifications in the specification language DAO.

The DAO environment is built on top of Interlisp-D (Interlisp with a graphics display)

```
((open RestaurantRoom open)
 (enter cgl RestaurantRooin)
 (seat wl cgl tabiel)
 (order cgl wl orderl)
 (bring wl Chef orderl)
 (cook Chef orderl chshSetl)
 (bring wl cgl dishSetl)
 (eat cgl dishSetl))
using old traces
 The reference for *ddMember/w1xqs5 is not found,
in the trace contained in contexts (T12)
originally non-existing state (wi.cgs * addMeffnber/wixgsS) not undone.
(Fail (Nullifying RestaurantRoom))
(Fail (Nullifying cg1»
(FaH (Nullifying orderl))
```

Figure 10.13: The Generated Trace and Analysis Result

and LOOPS [Bobrow and Stefik][4]. In normal use, these systems are intended to be transparent to the user. But the user is assumed to be familiar with basic concepts in graphic interfaces (e.g., windows, menus, etc) and, to have some knowledge of the concrete DAO syntax.

In the following we will first introduce the reader to the basic stnicture of the **DAO** environment. The functional aspects are presented afterwards. The scenario described at the beginning of this chapter can serve as a concrete example.

10.3.1 Basic **Structure of the User Interface**

The user interacts with the environment through several screen areas, each of which is a window or icon. An icon or a graph node in a window is associated with a group of operations that can be invoked by clicking mouse buttons which select a relevant item in a menu.

The menu items can be of two kinds. They can be commands, or DAO objects. The latter are the internal entities of the DAO system, e.g., representation of a relation class or the name of particular context (cf. the chapter on the internal representation). A selected DAO object is often an argument to a command.

^References for Interlisp-D, loops and DAO syntax are in [Xerox],[Bobrow and Stefik] and the chapter on DAO language respectively.

Our design favored windows with fixed positions and sizes (This refers to the time when DAO sets them up automatically. A user can always make changes directly using Interlisp functions.). The menus are "pop up" menus. We maintain that these are more appropriate for experienced users.

A. Windows in the DAO Environment

1. DAO Icon

   DAO icon is positioned at the upper left corner of the screen. Referring to the figure 10.2, the icon has the shape of "Taichi," the symbol of TAOism[5]. The commands included in the DAO Icon menu deal with global operations such as displaying the top level project map[6].

2. Status Window

   The status window provides information regarding the current status of the environment and the user's attention. Displayed therein are the release dates and version numbers of the DAO environment, Interlisp-D and LOOPS. Also displayed are the file directory currently connected to, the current project, and the current group. Referring to Fig 10.2, this is the window to the right of the DAO icon.

3. Browser Window

   The browser window is the largest window in Fig. 10.2. It uses tree and lattice structures to display projects, groups in a project, and definitions in a group. The nodes in the graph displays can be used as menu items from which the user can select a particular project, group, or definition to operate on. When a node is selected, an appropriate menu for the type of the node appears. Some of the operations in the menu change the objects themselves, others change what is being displayed in the browser window, and others affect the contents of other windows.

   The major function of this window is to provide a structure for working on specifications at the definition level--the most basic and frequently used level.

   A menu containing all the categories of the definition classes in the DAO syntax appears on top of the window[7]. This menu is called the definition class menu. A user

[5]By the way, "TAO" is spelled "DAO" in modern Chinese spelling.
[6]The top level project map is the data structure that contains all the projects and the subprojects and groups in each project. A user starts a session by choosing one of existing projects from the map or creating a new project as the Current Project.
[7]This menu is housed in a window especially created for it.

can choose to display and work on a particular category of definitions or all of them.

4. Display Window

   The display window occupies the upper right part of the screnn. It displays the contents of a definition or parts of a definition in a textual form or graphic form. It also shows the parse trees in list or graphic forms. The parse tree may have resolved references. Finally, this is the place where the simulator shows simulation traces.

5. History Graph Window and Primitive Predicate Window

   The history graph is like a flow chart for the events and temporal relations in the system history. It is drawn in a separate window. The primitive predicate window shows a definition for a primitive event or relation used in the history. Both windows are positioned at the lower right corner of the screen. The operations associated with the two windows will be explained in a special section below.

6. Recent Window

   The recent window keeps track of user interactions. When a definition is referenced the first time, its title will appear in the window. An entry in this window has the same operation menu as a definition node in the browser window. Because the entries are in historical order, this provides user with a convenient way to modify previous changes.

7. Class Menu Window and User Break Menu Window

   Their sole purpose is housing corresponding menus in them, which will be explained shortly. It is the window with title "Tree Menu" in Fig. 10.2.

8. Prompt window

   This is a standard window in Interlisp-D. Most of the time it presents system messages to the user. We also use it for accepting user input such as new project names, new group names or new directory names. It is positioned to the right of the status window.

9. Edit Window

   This is another standard window in Interlisp-D. The user edits definitions in this window. When a definition is newly created, the user will get a skeleton of the type of definitions he intends to write, and editing takes place on this skeleton.

10. TTY window

    This is the ordinary top level Lisp window. It is currently used as an additional space for displaying specification constructs and to report input/output status information. If every thing goes well one may not even need to type anything into this window at **all.**

B. Menus in the DAO Environment

In the DAO environment a menu can be associated with a window or with a syntactic class in the DAO language, e.g., projects, groups or definitions. In the latter case the same menu will pop up for the same object no matter where the object happens to be.

A complete map of menu functions in the DAO environment (in the form of trees) is available on line with explanations (you click each node in the trees to get the explanation for the menu item).

More than one menu can appear in each window. Very often a menu can be associated with another menu, too. That is, when one buttons a menu item, the effect may be the popping up of another menu. Ultimately, of course, some operation will be carried out for a menu item.

In the future we will refer to a menu by its associations. E.g., "Definition left menu" refers to the menu popped up when selecting a definition node with left button down, whereas "edit parse menu" refers to the menu obtained by selecting item "parse" in "edit menu."

In the DAO environment, the following conventions have been followed in associating the menus with windows or DAO objects and associating operations with menus:

Buttoning the right mouse key will give the user the standard Interlisp-D menu for the window.

With the left mouse key down (but shift key up), the user will generally get a set of standard functions specific for the selected item. For example, when a definition is selected in the browser window, the menu that pops up afterwards will contain items for displaying, parsing, simulating, and editing the definition.

With the left mouse key down and the shift key down also, a set of standard LOOPS operations will be available. These include inspecting LOOPS object contents and sending messages.

There arc also fixed menus. The definition class menu contains one item for each definition category. Buttoning with the left key will cause the definitions in the corresponding category to be displayed. The middle button is used for creating a new definition in that category.

The Edit Menu has four items: Parse, Done, Abort and Suspend. The user break menu is sometimes buried under the browser window. You bring it up by selecting break in the edit-parse menu. Their functions will be explained in the next subsection.

10.3.2 The Menu Operations in the DAO environment

In the DAO environment, the user requests an operation by selecting menu items from a menu or selecting a node from a displayed graph. In general, one chooses a menu item as a "command" and a node or a menu item as the argument for the command.

Consequently, a user is no longer bothered by the exact format and applicability of a command. However, he should know how and where to get the right menu or display.

Unlike terminal interactions which are recorded in the Lisp history list, the menu function calls can be found nowhere in the underlying Lisp system. The recent window plays a similar role as the Lisp "history," but it slows down the system considerably and we often turn it off.

A more serious problem with undoing is to erase things from the knowledge base when a definition is edited. This involves not only compiling the new definition, but also removing the class defined in the old one (probably invalid classes defined on top of it, e.g., events involving it or collections which are sets of objects of this kind.) and unassert axioms in the old definition. There are functions for doing this automatically.

A. Top level operations

The following top level operations can be performed by choosing an item from the DAO icon menu:

1. Load the top level project map.

   This operation is seldom necessary as the map will be automatically loaded when the user first enters the DAO environment. When it is used in the middle of a session, any modification to the top level, i.e., any addition or deletion of projects or groups would be destroyed.[8]

2. Write the top level project map.

[8]This is true when treating them as logical entities. Physically, the files that are on disk or definitions that are in core are all still there.

This operation is also seldom necessary as the map will be automatically updated when the user leaves the environment "decently," i.e. doing a "cleanup" first.

3. Display top level map.

   The map will be displayed in the browser window.

4. create a new project.

   One types the name of the new project into the prompt window. The environment will make a note of this and make the new project the current project.

5. Clean up.

   The modified specifications and Lisp files are listed in menu item form. They can be selected and saved.

6. Get help.

   (a). Help with commands. Presently the help provided is giving the user a tree of commands. A brief description of the effect of the command can be obtained from the menu associated with the command.

   (b). Help with syntax. The tree of DAO syntax classes will be shown. The syntactic rule associated with a class will be shown in list or graph form upon request.

7. Load subsystems.

   Some systems are not frequently used but take up much space, e.g., the parser generator functions. They can be loaded when needed.

Some top level operations can also be invoked by the commands in the header window, for example, resetting the connected directory.

B. Operations on Projects and Groups

In the DAO environment there is always a default "current" object at each level - a directory, a project, a group, or a definition. They are set explicitly or implicitly by some commands,. and other commands assume them as the relevant contexts (e.g., a new definition will always be created as a member of the current group, the command to display the graph of groups will assume the current project, etc.).

There are many similarities between the operations on projects and groups, e.g., the process for creating new instances of them.

The operations applicable to both projects and groups include:

1. Making a project or **group current.**

2. Deleting a project or group.

3. Moving one subproject or group from its current parent project to another parent project.

4. Renaming a project or **group.**

5. Creating a new subproject or group with the selected project as parent project.

6. Displaying all the subprojects or groups in the project.

The operations specific to groups include:

1. Loading the definitions in the group (which means loading the file that contains the definitions). This in turn will destroy the in-core contents of the definitions in the group.

2. Saving the group, i.e. all the definitions in the group, to a file.

3. Printing a hardcopy or "read form" of the group. (The syntax for definitions requires that the keywords should be present even if there is no specific content. A "read form" eliminates this and makes the specification much more readable.)

4. Displaying all the definitions or one category of definition (e.g., all the object class definitions) in the group.

5. Undeleting a definition in the group.

6. Parsing all the definitions in the group.

7. Compiling all types in the group.

8. Type checking and resolving references for all definitions in the group.

9. Checking the usage of types defined in the group.

C. Operations on Definitions

Definitions are the central and essential part of a specification because they contain the concrete information about the real world system. The operations on them are therefore numerous and we will present them in four categories: operations for viewing, editing, moving, and syntactically and semantically processing, definitions.

Most operations are invoked by choosing items from the menu associated with the definition nodes which are displayed in browser window. Some operations which involve editing and debugging definitions, are invoked from other menus. In the following if the latter is the case we will specifically mention it.

Operations for Viewing Definitions

1. Displaying a definition as a Lisp function, which is its internal representation, in the display window. All the comments (in the form of comments in the Lisp function) will be present.

2. Printing a definition as a Lisp function in the TTY Window. No comments will be visible.

3. Displaying a definition in the form of a normal parse tree, i.e., a tree with all the intermediate nodes being instances of DAO syntactic classes and the actual text as terminal nodes.

4. Displaying the summaries of an object (class)

   Presently a user can request a summary about an object class. This summary provides the collection of all the predicates, relations and activities that involve this object class. If a local summary is requested, only the definitions that directly involve this object class are listed. If a global summary is requested then the definitions that involve its generalizations will be listed, as well. For example, if you have defined a person to have an attribute "name," and a student is a person, then the summary of the student class will list "name" as a function applicable to a student.

5. Redisplaying the graph in the browser window.

   Note that this is not a user invoked function. When an editing session is finished it will cause the browser window to redisplay automatically. What is displayed is a graph for the definitions that are in the same definition category as the just edited one. For example, if an object class definition is just edited or created, all the object class definitions will be displayed.

Operations for Editing Definitions

1. Editing a definition.

   The content of the definition is brought to the edit window. The user modifies the definition with Standard Interlisp-D editors.

2. Creating a definition from a template.

   The user chooses a definition category (for example, an AgentClass or ObjClass) and the respective prototype definition will appear in the edit window. The user will be freed of the trouble of typing keywords and possibly making spelling or syntactic errors.

3. Creating a definition by copying an existing definition.

   The user can create a definition by modifying a similar one.

4. Suspending and resuming the editing of a definition.

   A user can temporarily leave the editor and suspend the editing by selecting the item "suspend" in the edit menu. He can resume the editing at a later time by selecting the item "edit: resume" in the left definition menu.

5. Moving a definition.

   These operations include all standard operations such as: deleting a definition, moving a definition from one group to another group and renaming a definition.

Operations for Processing Definitions

In this category are the operations for parsing, type checking or syntactic debugging of definitions.

The operations associated with definition nodes are:

1. Parsing a definition.

   A parser based on the concrete DAO syntax will parse the definition.

2. Displaying the parse tree of a definition.

   The parse tree can be shown in list form or graph form. The parse tree can be a top level parse tree (i.e., only the top level nodes are shown and they can be expanded later), a full tree or a partial parse tree (when the parse fails).

3. Displaying the corresponding syntactic rule for a syntactic class in graph form.

4. Entering the type defined in the definition.

5. Removing the type defined in the definition.

6. Type checking and resolving references for the definition.

7. Simulating the definition.

One of the most frequent activities of the user is to debug syntactically incorrect definitions. For this purpose, a particular package of functions is provided for his convenience. They can be invoked by first buttoning the item "Parse" in the edit menu which will pop up another menu in turn (we call it the "edit parse menu").

A definition should be made (explicitly or implicitly) "Current" for these functions to work. All these functions will take this Current Definition as their argument.

These functions permit the user to do the following:

1. Breaking Parser Functions

   For each syntax class there is a parser function associated with it. For example, for the syntax class "simple type" there is a function "SimpleType*" that can parse an expression that is a "simple type" specification.

   In debugging one's specifications one would often like to examine a construct at a lower level and possibly modify it. Breaking at a specific parser function enables the user to do this.

   The breaking facility in the DAO environment will always collect the parser functions immediately below the level of the current syntactic node for the user. For example, if you are at a break for "simple type" and you request to break again, the environment will recommend that the class "assertion" be broken. The reason is that the syntactic rule for "simple type" is

   SimpleType => [<article>] <ObjectName> suchThat ($Assertion) *.

   For the convenience of the user, one is also given the choice to specify any function name for breaking.

2. Unbreaking parser functions

   Conversely, the user can unbreak parser functions or whichever function that is broken.

3. Examining, Editing and Parsing Current Expression

   When a parser function is broken, the expression it is supposed to work on is designated in the DAO environment as "Current Expression." This expression can be displayed, edited, parsed and saved to a temporary buffer.

D. Operations on the History Graph and its Associated Knowledge Base

Sophisticated facilities are provided for interacting with simulation and analysis processes. Through menu operations the user can invoke various processes, set breaks, inspect how the inference is carried out and examine the state of the knowledge base.

Because the arguments of predicates are represented internally as LOOPS objects, which are scarcely readable, the system creates readable unique names (called "uname") for them. These names are not necessarily the original strings in the specifications because the same names in different scopes may stand for different identities. This is particularly true with quantified variables.

The user interactions are in the following categories:

1. Top level instructions.

   The user can select menu items to *draw the history graph* for a current definition, *start simulation*, ask for *traces* when the simulation is finished and start *analysis*.

2. Top level operations on history and knowledge base.

   At any moment of simulation, the user can request the system to show *the facts and rules in the current context*, all active contexts (i.e., all contexts included in the current context), the list of resolved functional references and the list of scoped variables (cf. DAO syntax). By "showing," we mean a new menu with associated operations will pop up and the displayed objects can be selected for further inspection. Many displays are in "concise form" where LOOPS objects are replaced with "unique names." They can also be simply printed on the screen which is faster.

   Deleted assertions are controlled at this level. They can be undeleted if desired.

3. Operations on contexts and assertions.

Once a submenu containing contexts is displayed, one can move around contexts by selecting desired ones. The assertions (facts and rules) can then be displayed. One can erase them or audit them (i.e., the theorem prover will show how the rules are used in proving).

4. Operations on nodes in the history graph.

   One can select a node in the history graph to inquire about the *facts with the same predicate*, the list of the *arguments*,[9] and the "unique names" of the arguments. For the latter two operations a menu will pop up and the user can examine the facts containing the argument. He can also set a new name for an argument for convenience. (e.g. when typing in a long formula for the system to prove.)

   One can also display the used definition of a predicate. This will appear in a smaller window created at the lower right corner of the history graph. The new window contains major parts of the definition as a set of menu items. The definition can be simply printed for reading, too.

   One can set a break at this node or unbreak a previously set break. The simulation will stop at the broken nodes.

5. Operations on "Used Definition" window menu items

   Most operations are on individual predicate formulas, so they are similar to the operations for nodes in the history graph. A particular feature is that at this point a user can ask the system to prove some relations, presumably a precondition for some event. Breaking can be set and undone at this level, also.

10.4 Experience with DAO

We have used DAO to produce several specifications.[10] The domains are business (a restaurant and a mail order book company), computer systems (a file system, LOOPS knowledge base management system and part of the DAO environment) and, other systems, e.g., a package router.

The writing of specifications often involves sorting out the concepts in the domain, or in other words, building a theory for the domain. Even with the help of a set of powerful tools it was time-consuming.

[9]Because the node label has to be adjusted to the size of a small box, the argument names are often truncated

[10]This section makes more sense if the reader waits until he has finished the last chapter

In our experience with composing specifications and analyzing them using DAO, there are good results, encouraging signs and revelations of limitations.

1. The framework is sufficiently expressive and imposes a useful structure.

   For the domain examples we looked at, we can express fairly conveniently what we want to say in the framework developed. There seldom are objects that we feel difficult in putting them into one of our categories. The temporal relations we have are sufficient in expressing various behavior patterns. Being able to talk about relative orders rather than using time variables makes representations concise and general.

   Our notations for expressing some functional relations are not rich enough to allow mathematical relations concisely specified but it can be done with clever effort.

   The preconditions and consequences of primitive events are not easy to specify correctly because we tend to either list too much or too little. But that often can be discovered later in the simulation.

   The framework enforces uniformity on specifications. We once put the precondition of seating as "not occupied by others." According to the specification scheme, since "occupied" is a variable property it has to be defined by some events. This made us realize that we should put "occupied" as the consequence of seating.[11]

2. The objective of checking out errors is achieved.

   In our experiment, some errors were intentionally left in specifications. For example, we reversed the order of events, we deleted one or two events or used wrong objects (with correct types). The errors were mostly detected. For example, if we did not specify the relation between dishesTakenAway and dishesCooked in some way, the simulator would stop and report a serious error. On the other hand, if we only specified in that dishesTakenAway was a subset of dishesCooked, the analyzer would point out that dishesCooked, being a noncomponent, was not completely nullified.

   More error discoveries came unexpectedly. For example, we specified that the dishes should be cooked by the chef and they should correspond to the order. But the waiter could not bring it to the customer because it was not mentioned that the waiter could access the dishes. Another time, there was a bug in the simulator and it did not run through to the end of the system history. In particular, the event "(leave

[11] Well, we could have realized it ourselves and we then do not need any automatic help. But "nobody is perfect."

customerGroup)" was not simulated. The analyzer complained that customer, as a noncomponent, should leave the system.

The power of some checking procedures came as a surprise. For example, the type usage checking discovered numerous things that we knew should be specified but failed to. That is, it not only found useless things but more often it found things that are useful but not used.

3. The system is reasonably sensitive to changes in specifications.

   We have noticed that the checking scheme is very rigid in type consistencies. This seems natural. But one is still surprised to find (bring (w: waiter) (eg: customer-Group) (x: item)) has a type error because the event "bring" is defined for (bring (pi: person) (p2 : person) (x: item)).[12]

   In simulation and analysis, the behavior change of the DAO system seems to depend on how much the specified event deviates from the regular history.[13] For example, if the event (enter (eg: customerGroup)) is repeated twice in the text, the analysis will report an error. But if (bring (w: waiter) (eg: customerGroup) (dishesBrought: setOf dish)) is repeated twice, nothing happens. In fact, this should be the case, because the waiter can walk back and forth twice to bring in the same set of dishes.

4. The various assumptions we made seem to be viable.

   For example, all the examples we specified or did preparations for specifying, can be treated as stable systems. As another example, in all the specifications we worked on, preconditions are all in simple forms. The fact that there is no single instance of disjunction of assertions is probably a coincidence, but this shows that the percentage is low and the rationale we gave is not groundless.

5. Disciplined specifications help deepen our understandings of systems.

   The framework we built is based on some general analysis of systems. Specifying systems in this framework turns out to be helpful in understanding particular systems. For example, in trying to define the object class of files, we used the notions of information object and physical form, etc. We noticed many layers of abstractions for files, e.g., as texts, as character strings, as bytes. We found that different events are

[12] We intentionally used customer group instead of customer to see how collection objects would respond to our kind of systems (Customers come in groups and pay in groups. So this is more realistic too), and we found our specification schemes have to pay special attention to this problem.

[13] This is just a comment. We do not have a measurement or a comparative study.

associated with particular levels of abstractions. The type checking for the restaurant example helped us to realize that the object "order" is used as a pure value when it is put in the consequence of event ordering:

    customerGroup.currentOrder «— order,

but an information object with identity in

    (bring (w: waiter) (chef: chef) order).

The analysis, being based on causal chaining, can suggest alternative system designs. For example, in the restaurant specification, after finding independent paths, we discover that paying and serving food are independent of each other after the ordering. Therefore, the two subpaths can be interleaved in any fashion.

Many intuitive questions from users can be put in theorem form and answered. For example, one may ask whether, in this restaurant, a customer can always eat what he has ordered, or whether some mixup may happen. Put them in formulas, i.e.,

    ((forAll x in order)(forSome y: dish)(&)

        (and(eat c: customer y)(y.name =x)))

meaning all he ordered is eaten by him and

    ((forAll x: dish)(if (eat customer x) then (in x.name order))

meaning what he eats is what he has ordered. The proof of the formulas is the answer.

6. The system shows acceptable performance:

All the features, algorithms mentioned in this report, unless otherwise stated, are implemented. Our system DAO is built on top of a modified version of the base level MRS (in terms of theorem prover) and the kernel of LOOPS. The latter is, in turn, built on top of Interlisp-D. The system can run on any Xerox Lisp machines. For the part of editing, browsing, and compilation, DAO performs fairly well on both Xerox 1108 and Xerox 1132. An example specification of the size of 6-7 pages can be parsed and compiled "without waiting". Because there are often complex theorem proving tasks in it, the simulation may get very slow on some occasions. This is partially due to the fact that we do not have inference control. To simulate a specification of a similar size, it will take about 10 minutes on a Xerox 1132 with 1.5-3 megabyte memory. But the performance is greatly improved if we use a machine with extended memory, say, with 8 megabyte core memory.

A specification of real size will take hundreds of pages. For this kind of problems, the present system performance is not satisfactorily practical. However, we note that this research has not aimed at producing a practically usable program. At this stage, it is the the validities of the ideas that matter. The aspect of performance should be paid attention to only to the extent that it is relevant in judging the plausibility of the ideas. Boosting the performance of the system for its own sake is too premature for our research now.

7. There are limitations.

   There is one fundamental limitation that has been mentioned many times. That is, the analysis only discovers violations of general domain constraints but not violations of system specific constraints. In any system, the dishes in the kitchen need someone to bring them to the dining table. On the other hand, if the waiter decides that he would do it twice or three times it is perfectly possible, and our checking can do nothing. This means that if the particular system really does not allow bringing to occur twice for a single course, our system can not detect the problem. (It may be revealed through some other violations of general constraints, though.)

   One severe limitation is that we can not embody as many discriminative categories as we want. Many theoretical results can not find uses. (cf. the section of the overview of the DAO language in Chapter 9) As a result, we do not specify the distinctions among the types of objects or relations to the extent that is allowed by our ontological analysis. For example, in the actual implementation, we do not process the category "social relations." They are treated the same as many other abstract relations. Moreover, we have some specifications that are superficially not consistent with our analysis. For example, in our restaurant example, the collective object, order, is an information object having a corresponding physical form, and, therefore, an identity. This is demonstrated in its being involved in events. As such, it should be more than a pure value. On the other hand, we know that a customer's "current order" is updated after each serving. Therefore, the value of this property is a pure value. However, for the consequence of the event ordering, we have a formula

   customer.currentOrder ← order.

   This inconsistency can be resolved in our actual processing system. It is done by assuming an object such as "order" will have two status in different contexts. The system itself is responsible for figuring it out.

   As the system stands now, the error messages are sometime spurious. There are many false alarms. And it needs user instructions when it can not decide.

Lastly since we have been working on small examples at very high abstraction levels, the research results can not be immediately applied to how the actual system development can be improved.

Chapter 11

The Internal Representation of DAO

11.1 A Hybrid Representation System

Frame-based vs. Proposition-based Representation Systems

The base level of our implementation is in Interlisp-D, a dialects of Lisp. To build a system as complex as ours, we need high level representation systems. These systems are mainly of two kinds: Frame-based or proposition-based. They are good for different representation tasks.

If we define a "relation space" by a set of matrices, whose rows are indexed by all possible instances of one type of relation and whose columns are indexed by objects being related, then some matrices would be very dense while others would be sparse. We may then talk about the relation space as being dense or sparse.

Frame-based systems suit well-structured domains, that is, when the relation spaces are dense ([Minsky][Bobrow and Winograd][Schank and Abelson]). Proposition-based systems suit sparse relation spaces or less structured domains [Genesereth et al.]. Because of this "structuredness," a frame-based system hints us as to where to look for potentially interesting properties. But for the same reason, it is rigid and can not readily accommodate new relations. A proposition-based system allows us to put anything at any time into the knowledge base. But the system is flat. Therefore, in choosing a frame-based system vs. proposition-based system one would be trading structure for flexibility.

Frame-based systems are mostly implemented in "slot-value" fashions. Proposition-based systems are often in the form of simple lists like logic formulas. This entails some other technically significant differences. Frame-based systems suit binary relations. Proposition-based systems support n-ary (n>2) relations. Frame-based systems suit single indexing (or at least there is a major index) whereas proposition-based systems multiindexing. Because the two forms of representation are different, the inference engines are different. In particular, frame-based systems often use procedural attachment, while general purpose theorem provers are easier to accommodate in proposition-based systems.

Combining Frame-based and Proposition-based Representations

We chose to use a hybrid representation system hoping to take advantage of the strength of both representation schemes. We chose LOOPS ([Bobrow and Stefik]) as our example of a frame-based representation and MRS ([Genesereth et al.]) as that of a proposition-based representation. For detailed descriptions of the two systems, one has to consult their manuals. Here, we just mention that LOOPS is both a language and an environment that combines data, object, and rule oriented programming. The attributes of an object are stored in instance variables (IV) and the attributes of a class are stored in slots named "class variables." . The invocations of procedures associated with object instances are accomplished through the so called "method" which is equivalent to "message passing" in most object-oriented programming languages. MRS, on the other hand, is a representation system designed for providing multiple representation media and flexible inference control. The data are mainly stored as propositions in list form and fully indexed by all elements in a list (with a few exceptions such as numbers).

A subset of LOOPS is used as the base language. Besides the generally attractive features of frame-based representation, in terms of implementation status, LOOPS is robust and has fairly good performance. Its data access is fast.[1] The data access scheme does some checking, also.[2] Its object-oriented programming style brings in some fine features, such as message passing. For example, the message passing scheme suits very well the writing of recursive descent parser functions.

However, we did not use all the available packages in LOOPS. For example, LOOPS' rule-based programming part. The LOOPS rulesets are basically production systems. As a reasoning system, they essentially have only universally quantified variables. The inference control is also too rigid.

We chose part of MRS as the proposition-based upper layer of the system. The propositions are in list forms. The heads of the lists, the predicate names, are Lisp atoms,

[1] There are concerns that all the IV values should be accessed by message passing, which LOOPS does not do for efficiency considerations.

[2] It checks to see if the IV is defined for which a putValue of getValue is called.

but the rest of the lists are all LOOPS objects. Every proposition that is originally in the specification, has a LOOPS object representation, too. Besides the common features of a general purpose theorem prover, the inference mechanism of MRS has a flexible and powerful inference control, which we have compiled into fast running Lisp functions.[3] :

Tradeoffs in Selecting Implementation Schemes

We can not just get all the advantages of both representation framework but avoid all shortcomings in our hybrid system. Which feature is to keep and which is to give up is a key problem in the design of the internal representation system of the DAO specification system. Here the keyword is "tradeoff."

In comparing designs of mechanisms there are two levels that one should be aware of. At the first level, the competence level, we need to know how much a mechanism will be able to do. At the second level, the performance level, we wonder how well it can do it, perhaps depending on the particular kind of tasks it is supposed to accomplish. The trade-off is not necessarily within one level. We may be willing to sacrifice some competence to boost the performance. If the lost generality is more than compensated by a drastic improvement in performance then it is worth considering. Here, it is important to know how the lost competence manifests itself. It is very undesirable if the mechanism simply produces an erroneous response. On the other hand, it would be satisfactory if the system knew that the case was beyond its ability and informed the user.

Tradeoffs in designing processing schemes arise when we have more than one plausible alternative. As we have explained earlier, this is the case with DAO since it uses both object-oriented representations and list representations. DAO uses LOOPS [Bobrow and Stefik] as the basis of its frame-based representation system. All parsed specifications are in the form of LOOPS objects. When a proposition has been represented in a compiled (i.e., type checked) LOOPS object form, but the user also wants to use its list form, in order to keep the information stored in the LOOPS object available,[4] there are many choices. One can, when translating the LOOPS object form into a list, hashlink the list to anything one likes, even the LOOPS object itself. One can do nothing at the outset but do a brute force search to find that LOOPS object back if there are not too many LOOPS objects (say, a few hundred). Another possibility is to redo the type checking for the list form of the

[3]There are some problems with the version of MRS which we use. They are all implementational in nature. For example, the theorem prover does not handle nested quantification forms, the "unassert" procedure only works for the local context, and so on.

[4]There are many reasons for desiring so. But there is no inconsistency problem, however. Those data recording changing states of described systems are in list forms only.

proposition! (Remember the original form of the specification is in list form!)

The decision on whether to recompute each time one needs it, or to store it somewhere, or to do a brute force search, can not be made a priori. It should be decided based on the data one wants to process. Before we have enough statistics on the actual data, we can just list all the alternatives and implement one of them that we consider (guess) to be most appropriate and hope for the best. This was our guide in designing our representation systems.

11.1.1 Organization of Data and Knowledge Base

Organization of Data Structures

The major data structures in DAO are stored in three forms : Lisp functions, LOOPS objects, and lists of propositions.

Each group of specifications is stored in a Lisp file. Each definition in a specification group is in the form of an Interlisp LAMBDA function. As a result, Interlisp facilities for function manipulation can be directly used for modification of definitions. The syntactic rules are in the form of IV values of LOOPS objects. But the parser is a huge collection of functions which is generated from the rules.

Parsed specifications are in the form of parse trees. The parse trees are represented as LOOPS objects. In terms of concrete syntax, a child node in the tree is linked to its parent node through CHILDREN and PARENT IVs. In terms of abstract syntax, a node has several nodes as attribute values, specific to its syntactic category. The non-terminal nodes are LOOPS objects, and the terminal nodes are the text strings. For example, a node of collection type has at least two IV values: "collectionTypeKey" and the base class (these are @collectionTypeKey and @type, respectively). A piece of specification, such as the part after ":" in (StudentGroup: setOf student), would be a node for a collection type, represented as a LOOPS object, say #&XYZ. It, in turn, has the slot @collectionTypeKey filled with the string "setOF," and @type, filled with the string "student."

For ease of access, the names of and pointers to the classes defined in specifications are stored as property lists for later type checking. This is done during compilation.

At the beginning, the references in a parse tree are not resolved. After the resolution is done, the tree for history is called the history tree. It is also a LOOPS object. A graphic representation, the history graph, is generated based on this history tree. It uses a modified interlisp-D graphics package.

Many axioms in specifications are time-independent assertions. Most of them are in the form of implications ("if then"). They are also called "rules". At first, when they are just parsed, they are LOOPS objects. Then they are transformed into list forms (a procedure called "listifying") and asserted in an appropriate context (see later subsection on the "context mechanism") in implication form or conjunctive normal form (CNF) according to the complexity of the axioms themselves (see a later subsection on simple form formulas).

In the simulation process, time-dependent assertions, that is, "facts" (by the way, as opposed to both rules and facts, a formula that is to be proved will be called a "query") are constantly asserted and unasserted. Some of them are originally in LOOPS object forms, but, eventually, they are all converted into list forms for processing. Like rules, they could be put in CNF form or implication form. But if they are the so called "simple form formulas" (see the later subsection on simple form formulas), they can be processed almost in their original form. This will be elaborated shortly.

Besides axioms, the other time-independent data, e.g., the set membership of an object and the hierarchy of classes, are always stored in LOOPS object form or object IV form. Special functions are called if this information is needed in proving propositions.

Finally, the traces, the basis of analysis, are lists of propositions whose arguments are LOOPS objects.

We note here that we differentiate clearly between LOOPS objects and objects in our domain model. The only place they are related is that a LOOPS object is used to represent an entity in the specification text and the entity may turn out to denote an object in the model. In fact, the properties of an object are seldom the IV values of its corresponding LOOPS object (exceptions are set membership, class, class hierarchy etc.). Furthermore, the messages ("method" in LOOPS term) are never related to our "events." Interestingly, our events are represented as LOOPS objects, originally. This makes it easy to treat events as terms in the sense of logic (cf. the chapter on temporal relations).

Partition of the **Knowledge Base**

One problem of a hybrid system is the partition of the knowledge base. In our particular system, as mentioned before, some knowledge is in the form of LOOPS data and some is in the form of lists of propositions. For example, the fact that "$stu" is a member of "class 85 of Stanford" may only be represented as a value "dass85Stanford" for an IV "schoolClass." To make use of this information in theorem proving, we need a special routine that extracts the information from this IV and interfaces with MRS. However, even with this special routine, we can only go from $stu to his class but not from the class to $stu. For example, we can not find all the students that are in the same class with $stu, if all the relevant information is similarly stored.

The special routine may be done away with since MRS allows the user to specify **how a** particular kind of inference should be done at meta level, in the form of (Myto <action> <afgs> <procedure>). In this case, (Myto Member $x '(GetValueOnly $x 'set))[5] may work. Still, that procedure can only guarantee that we find all y's satisfying (member $x y), but there is no way for it to find all x's given the predicate "member" and the supposed set y-

Because the proofs of many relations and properties depend on linking the object in question to the right class or superclass and the right set or superset, the influence of the above problem could be quite severe. It would leave us unable to prove things that could be otherwise proved, because in a particular reasoning mode the system simply can not make use of its knowledge.

Fortunately, for all concrete object instances, e.g., John, Mary or Joe, their set memberships are never established in this way. As we pointed out, "our" objects (the objects defined in DAO specifications) do not keep their properties in IV values. The relations in the text are, at first, parsed into the form of LOOPS objects and then converted into lists of propositions. Only in the case of quantified variables, do the two schemes intersect[6] and this kind of problem arises.

But the class instanceship of John *is* represented in IV value form. If John is in the class of "person," working on (class $x person) may not find John for us. We note that this and similar cases, i.e., collecting all instances of a class, are fairly rare. On the other hand, queries the other way around are much more common.

Here we have a tradeoff. For all the relations that are mentioned above we could require that their proposition counterparts be asserted into knowledge base, and thus guarantee some completeness. But the question is that, for the particular kind of problem at hand, this only complicates processing by cluttering storage, but does not add much problem solving capability.

11*1.2. The Use of Typed Quantification Variables

A positive outcome of the marriage of two kinds of representations is that we can use "typed quantification variable'[1] and the formulas containing them.

In the DAO language, the variables in quantifications can be plain variables, vari-

[5]This may be not the exact MRS format. We may need a special function for the <procedure> part.
[6]Interestingly, this is because the variables are both "our objects" and formal entities associated with quantifications.

ables with a type (followed by ": <type>") or variables with a set (followed by "in <collectionObject>") (cf. DAO syntax). The latter two are generically called "typed variables". They are further divided into "class variable" (not LOOPS class variable!) and "set variable," respectively.

The explicit forms of quantification (forAll x: X) and (forAll x in S) can sometimes be equivalently denoted by $x@X or $x@S where X is a class and S a set. We similarly have ?x@X and ?x@S for (forSome x: X) and (forSome x in S). These variables are all internally represented in the form of LOOPS objects. The set or class information is actually stored as IV values. A formula using these variables to represent quantifications is called an *implicit quantified formula.* In the display of the actual system and in future discussions we will use implicit quantified formulas whenever possible. First we will give a justification for that.

This form of writing formulas has a kind of conciseness (and therefore, clarity), because we can avoid explicit quantifications and implications. For example, before we had to say

(forAll x)(if (member x DishesCooked) then (at x Customer)),

or

(forAll x in DishesCooked) (at x Customer).

But we now only see

(at $x@DishesCooked Customer).

Similarly a rule asserting "if a set of objects S is at place x then each element in S is at x" can be represented as

(if (at S $x) then (at $y@S $x))

But this is not essential as many readable forms can be designed which do not have any processing advantages. The power of this representation is that we can use pattern matching to handle this implication rather than use resolution method, which is computationally much more expensive.

To be able to match typed variables, the ordinary pattern matcher has to be modified substantially. Attention should be paid both to correctness and efficiency.

The procedure of this generalized pattern matching is:

1. Check if the types agree. This means to trace class and superclass chains as described in the chapter on compilation.

2. If the type checking fails then fail. If it succeeds and the variable is a class variable then succeed.

3. Check if set membership or set inclusion holds. This implies possibly tracing super set links.

4. Succeed if the checking succeeds. Otherwise fail.

Because the checking uses very efficient functions to check class instanceship and set membership, and because the proving is done in pattern matching, the theorem proving can be very efficient.

11.2 A Special Purpose Theorem Prover

11.2.1 Simple Form Formulas and Pattern Matching

The power of typed variables manifests itself even more in the use of "simple form formulas" – a concept proven to be very handy for our particular problem solving tasks.

The notion of simple form formulas comes from our specification practice. We find that most preconditions and consequences of (primitive) events are not quantified formulas. Even when they are, the quantifications are seldom nested. On the other hand, the events are never quantified over classes or over plain variables (i.e., variables that are neither class variables nor set variables). That is, they can be only quantified over a set. Most of the time, events are not quantified at all.

When an event is quantified, it may be quantified over more than one variable. For example, we may have

(forAll x in order)(&)(cook Chef x (d:dish)).

As we explained in the chapter on DAO language, there is an implicit existential variable. That is,

(forAll x in order)(forSome d: dish)(&)(cook Chef x d).

It turns out that the above example shows a very frequent pattern in specifications. There are no arbitrary nestings. The events happen to some collection objects and their "corresponding" collection objects, which makes a pattern of "∀x ∃y e(x,y)". Because of this pattern of events, it follows that if the precondition for event e(x,y) is p(x,y), then one is likely to be proving a lot of formulas of the pattern of ∀x ∃y p(x,y).

The other side of our interest on this pattern is that *it can be handled by pattern matching*. As we mentioned before, pattern matching is a very efficient processing scheme. Moreover, an explanation generated from a reasoning process based on pattern matching (e.g., backward chaining) is more understandable than that from a scheme based on resolution.[7]

Since there is both the need and the possibility, the introduction of this special form of formulas is well motivated.

[7]There are systems that generate readable explanations from resolutions. But they themselves are complex systems. Our system can not afford that.

Now one may wonder where those complex nestings of quantifications are. They do not disappear. They are mostly in rules and this is where they have always been.[8] Of course, there can exist arbitrarily complex formulas anywhere. Our point is simply that, statistically, they seldom enter into specifications of real world systems.

Definition of Simple Form Formulas

A generalized atomic formula is defined to be of the form

$$\forall x,y\ldots\exists u,v,\ldots[[x\in X \land y\in Y\ldots]\rightarrow[u\in U \land v\in V \land p(x,y,\ldots u,v\ldots)]],$$

or alternatively

$$\forall x\in X,y\in Y\ldots\exists u\in U,v\in V,\ldots\ p(x,y,\ldots u,v\ldots),$$

or

$$\forall x\in X,y\in Y\ldots\ p(x,y,\ldots f(x,y\ldots),g(x,y\ldots)\ldots)$$

where p(x,y,...u,v...) is an atomic formula and f and g are skolem functions.

Its implicit form is written as

(p \$x@X \$y@Y...?u@U ?v@V...)

or

(p \$x@X \$y@Y...f(\$x@X,\$y@Y...) g(\$x@X,\$y@Y...)...)

in our notation. If we do not emphasize the exact sets, we may simply write \$x@ or ?u@.

A generalized literal is a generalized atomic formula or the same quantification with the atomic formula p(x,y,...u,v...) negated. The negations can not be simply written in the form of

NOT (p \$x@X \$y@Y...?u@U ?v@V...).

This is because if "NOT" is to mean "¬" then this would mean

$$\exists x,y\ldots\forall u,v,\ldots\neg[[x\in X \land y\in Y\ldots]\rightarrow[u\in U \land v\in V \land p(x,y,\ldots u,v\ldots)]],$$

which can not be a generalized literal as we have defined.

A *simple form formula* is the conjunction or disjunction of generalized literals.

Using the axiom schema in natural deduction ([Manna])

$$\forall x\ P(x)\land Q(x) \equiv \forall x\ P(x) \land \forall x Q(x),$$

we can easily show that the events or relations connected by temporal connectives ";", "!" and "&" are simple form formulas if their components are.

We now explain why negations are handled in our scheme in an unusual way. This follows from the way the pattern matcher works, for which our simple form is introduced.

A pattern matcher for handling literals is simply a unification procedure ([Manna]). It

[8]Another observation: Rules have only class variables.

can handle universal variables and skolemized items (which are equivalent to existential variables). If the unification (or matching) is successful, a list of bindings will be returned.

From the way bindings are produced we can make the first corollary. That is, if we have a formula in the form of

(p \$x@X \$y@Y...?u@U ?v@V...),

it will assume the weaker interpretation, i.e., universal variables precede existential ones.

In standard procedures, such a unification is always done after moving the negation symbols all the way inside. From this we have the second corollary. That is, the types of quantification (being universal or existential) are assumed to have been already inverted if an inversion is ever needed. This means that putting "NOT" before

(p \$x@X \$y@Y...?u@U ?v@V...),

i.e.,

"NOT" (p \$x@X \$y@Y...?u@U ?v@V...)

would be taken as

$\forall x \in X, y \in Y\ \exists u \in U, v \in V\ [\neg p(x,y,\ldots u,v\ldots)]$,

by the pattern matcher–unifier.

Given the meaning of the notations we have introduced, we can not simply write "NOT" before a generalized atomic formula to see how the pattern matcher works. To put it another way, we can not see how a generalized literal with negation will look in our notation. We introduce a new negation symbol NOT_s, meaning "special not." It "pulls" the quantifiers in the generalized atomic formula before itself. So

NOT_s(p \$x@X \$y@Y...?u@U ?v@V...)

is equivalent to

$\forall x \in X, y \in Y\ \exists u \in U, v \in V\ [\neg p(x,y,\ldots u,v\ldots)]$.

This is the generalized literal with negation. We remark that in this way the pattern matcher never sees "$\neg$" now, it only sees "NOT_s". [9]

Clearly this notion of atomic formulas can not handle many quantified formulas. Typically,

$\neg\forall x\ \exists y\ p(x,y)$ (equivalently, $\exists x \forall y\ \neg p(x,y)$)

can not be handled. However the reader can verify that if there is only one type of quantification (only universal or only existential), all combinations of negations and quantifications can be handled. For example,

$\neg\exists x\ \neg p(x)$

[9] In the actual system or code, the same string "NOT" is used. Only when it enters the matcher, is its meaning changed.

can be handled since it is equivalent to $\forall x\ p(x)$. [10]

Preconditions in Simple Form

In explaining the motivations, we mentioned that preconditions for primitive events and conditions for a quantified event are mostly simple form formulas. We will now take a closer look. To make our presentation simpler, in the rest of this chapter we will refer to the conditions necessary for a quantified event as "precondition", as well. This kind of precondition is referred as "combined precondition" sometimes.

First we mention that a precondition that is to be proven for a quantified event has two parts. The first and obvious part is in the definitions of the individual primitive events in its body, which is the usual case. But there is another part: the quantifications. For example, the precondition for a waiter bringing an item x to a customer c may be that there is no other customer using x. If we are talking about a waiter bringing a collection of things S to c then the precondition should be

$\forall x \in S\ [\neg\exists y\text{:customer}[\text{occupy } y\ x]]$.

The effect of the second part would be even more impressive if we have an event of the form "$\forall x\ \exists y\ e(x,y)$", but the precondition for $e(x,y)$ is just $p(y)$. In this case, the set containing x will affect the proof of the precondition even if it is not present in $p(y)$. For example, if we are to find a hard copy for each file in a directory and the hardcopy has to be readable, i.e.,

```
        (forAll file in filesInDirectory)(forSome hc: hardcopy)
             (and (find file hc )(hardcopyOf file hc)(readable hc))
```

or, assuming the set of hardcopies are identified as HC (cf. the chapter on DAO language),

```
        (and (find $file@filesInDirectory ?hc@HC )
          (hardcopyOf $file@filesInDirectory ?hc@HC)(readable ?hc@HC)),
```

then we can not simply prove that there exists one readable hardcopy. We have to prove that all the hardcopies are readable since, in general, for e($x@,?y@) ?y is essentially f($x) and $x is universally quantified.

Summarizing this, we mention an axiom schema in natural deduction which will be a basic axiom in our discussion:

$[\forall x\ (p(x) \rightarrow q(x))] \rightarrow [\forall x\ p(x) \rightarrow \forall x\ q(x)]$.

This can be easily extended into

$[\forall x\ \exists y\ (p(x,y) \rightarrow q(x,y))] \rightarrow$

[10] Whether this is done by the user or a transformation routine is a separate question.

[Vx 3y p(x,y)->Vx3y q(x,y)]

and

[Vx€X 3yGY (p(x,y)^q(x,y))] ->

[Vx€X 3y€Y p(x,y)->VxSX3yeY q(x,y)J.

Of course the inverse is not true.

These are the basis for our combining the two parts to make the actual preconditions for quantified events.

Handling the case of a precondition without "the second part," i.e., the quantification from a quantified event, is simple. The definition of simple form formula can be directly used. Namely, when there are quantifications in the precondition of a primitive event, if the quantified formula is in simple form then the precondition is in simple form. If there is a negation, push it all the way inwards and decide. If it is a conjunction of simple form formulas it is a simple formula.

When a combined precondition for an event does have quantifications in the form of "V...3..." there are the following cases of the first part, i.e., the part associated with the precondition of the primitive event:

1. Quantified formula

   In this case, the quantification in the primitive event (after moving negation all the way inwards) should be existential only. The formula body is subject to other constraints discussed below. This means that some of the resulting combined preconditions would not be in simple form. For example, one may specify the precondition for "bringing something to customer" as "everybody likes it," i.e., if the event signature is

   (bring u: waiter v:customer y: item)

   the precondition is

   (forAll x: person) (like x y).

   The combined form will have a quantification in the form of (forAll ...)(forSome ...)(forAU).

2. Negation and conjunction

   If negation is on (generalized) atomic formulas, it is all right. Note that in this case, the negation is automatically a *NOT_8*. Actually, this is what motivated our definition.

   As our first basic axiom shows, any conjunction at any level (after moving negations) is all right.

3. Disjunction and implication (after moving negations all the way inward)

   The resulting formula is not a simple form formula.

4. Formulas with missing variables

   This is orthogonal to the other cases. Recall the example in the beginning of this section: a missing universal variable should be "restored" to the existential variables that are affected by it. Usually this amounts to changing a formula of the form

   (p ?y@Y ?z@Z),

   to, for example,

   (p $y@Y ?z@Z).

The above discussion provides a set of criteria for deciding if a formula is in simple form. In this system, every formula has to be checked for this before it is sent to pattern matching or resolution.

We note that not only the cases covered by this form of formulas are limited but also the proof procedure, pattern matching itself, is not a complete deduction system. Pattern matching is actually based on one of the deduction schemata in natural deduction, *modes ponens*. Therefore, many other natural deduction schemata are not included. We have included in our system some of them in procedural form, for example

$$P \rightarrow [P \text{ OR } Q]$$

or

$$\text{NOT}(\text{NOT } P) \rightarrow P.$$

However, in the case of

$$(\exists x((p\ x) \vee (q\ x))) \tag{10.5}$$

and

$$(\exists x(p\ x)) \vee (\exists x\ (q\ x), \tag{10.6}$$

although (10.5) and (10.6) are equivalent, the proof of one can not be reduced to the proof of the other in our system.

Multiple Levels in Proving Simple Form Formulas

Our representation system has both advantages and shortcomings in carrying out efficient proofs. The LOOPS objects can hold much information but type checking for objects may be expensive. Our proof procedure tries to minimize type checking operations and extensive search. It does this by conducting the proof in mutually supplementary steps. This is summarized in picture 11.1.

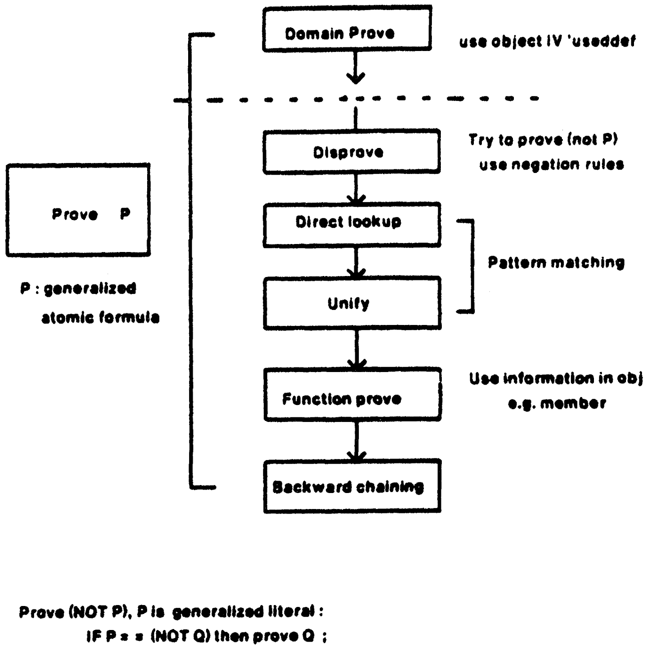


Figure 11.1: Steps in Proving a Generalized Literal

We explain each step in turn.

1. Prove directly from LOOPS objects

   Predicates are in LOOPS object form after they are parsed. They are then type checked. This type checking can resolve the references of the predicates, too. Namely, if you have two predicates defined for different classes of objects but they have the same predicate name, the type checking will notice this and link each predicate to the correct definition. The linking information is stored on the LOOPS object for the predicate references.

   The list form resulting from "listifying" the LOOPS object will not have this information. Therefore, we may want to do some proving just before "listifying." For example, if the axioms for proving someone's owing money to a person and his owing money to a bank are different *and* some of these axioms are proceduralized, we may want to let the procedure to work on the LOOPS object first.

   This step has many limitations. For one thing, it works only for those queries that do not need return a binding, i.e., they only need a "yes-no" answer. The problem here is that if we are proving a conjunction of formulas containing the same variables then the bindings obtained in proving each individual formula have to be checked for consistency. But to keep the binding obtained before "listifying" and to send it to the right place for comparisons increases the complexity greatly.

2. Disprove

   This step is easy to explain. Sometimes it is easier to prove the opposite of a query than to exhaust a whole list of possibilities. Special negation rules in the form of "if ...then (NOT ...)" have to be prepared separately by specification writers. They are used at this step.

3. Pattern Matching

   This includes two substeps. The direct lookup will check if an assertion in exactly the same form as the query is present in the knowledge base. If it fails, it will use generalized pattern matching.

4. Prove using functions

   Special functions are necessary in proving the properties represented as relations between LOOPS objects, for example, set membership (cf. the section on partition of knowledge base).

5. Backward chaining

   Backward chaining is the last resort in proving generalized literals using pattern matching.

11.2.2 Proving Formulas in Both Simple and Complex Forms

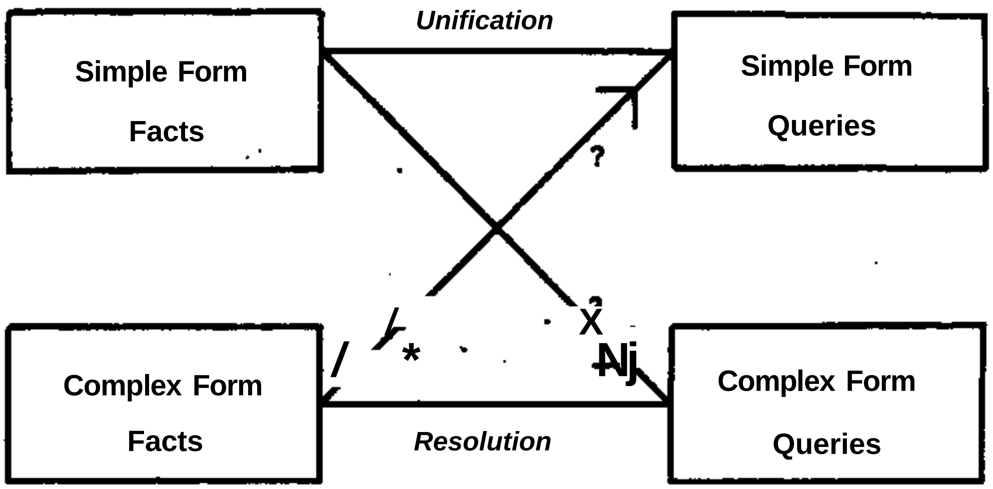
In our kind of system we use facts and rules to prove queries. Preconditions are the most frequent queries. As we pointed out, they are often in simple form. But we note that they can also be in non simple forms which we call *complex forms.*

If our proving routine for simple form formulas is successful in 95% of the cases, we still need some way to handle the other 5%. This has to be dealt with through a general theorem proving procedure. In our case, we chose to use resolution method. It uses the linear support strategy. Figure 11.2 shows how this is done.

We try to prove queries from facts and any one of them can be in simple form or complex form, so we have four cases.

1. Simple form fact to simple form query

   This is the case where we use pattern matching.



Simple FormFacts = > Complex FormQueries

: reassert facts in complex form

Complex Form Facts => Simple Form Queries

: starting from the point, use resolution

Figure 11.2: Handling Different Forms of Formulas

2. Simple form fact to complex form query

   The query can only be handled by resolution and the formulas have to be in explicitly quantified form. Since all the simple form facts have been asserted in the knowledge base in implicit quantified form, we need to reassert all the formulas that may be relevant in explicit quantified form. An alternative is keeping two representations all the time.

3. Complex form fact to simple form query

   The problem becomes complicated when we have a fact in complex form.[11] This is because once you have a fact of this form you never know if it will be needed in the proof of a query. Therefore, this requires that we use all explicit quantified forms and *from now on* use resolution only. Of course, if the query is in implicit quantified form, it is to be converted, too.

4. Complex form fact to complex form query

   This is similar to the last one. All the representations would have to be converted into explicit quantification and the ensuing proof uses the resolution method.

The above two modes of deduction suggest the use of two representations at the same time. For pattern matching, implicit quantified formulas are to be used. For resolution, conjunctive normal form would be ideal. Of course, ordinary quantified formulas are neutral but they do not provide processing convenience for either of the deduction modes. Unlike the hybrid representation case, this does not partition the knowledge base since we are employing duplicates. However, there may be data integrity problems. That is, we have to update all the representations for the same facts once there is a change. One solution to this is to use hash links to chain the two forms together so that they can be updated simultaneously.

11.2.3 Context Management

The notion of locality of causation offers some potential help in handling the frame problem ([McCarthy 1977]), but this direction has not been explored in this research. Currently, the frame problem is dealt with by simply writing frame axioms for operations.

In practice, our simulation scheme handles changing situations through a context mechanism. It is directly based on the theory mechanism implemented in MRS. Every time

[11] I did not find a natural example of this kind in writing specifications.

a conditional exists, i.e., two possible traces exist, the context splits. The newly formed contexts would be the children of the original context. The assertions in the parent context will remain true for the trace associated with either of the new contexts unless explicitly erased.

We note that this context mechanism is not efficient enough, noticing that many events in the two traces may be exactly the same. However, it is favored because of its simplicity.

A complexity coming from this context mechanism is that, to unassert an assertion, one can not simply come to the current context and do an "erase." For example, if you are unasserting a fact P for context c but the fact is physically included in context C which has children c, c', c", ... you need to reassert P for all other contexts, and only then you can erase P for C.

Chapter 12

The Compilation of DAO Specifications

12.1 The Role of Compilation in Specification Validation

The DAO compiler plays a similar role to the one played by compilers for more traditional languages. Namely, it parses the text, reports syntactic errors, builds a parse tree when the parsing is successful, type-checks the arguments for a composed structure and links a reference to its definition (for types) or declaration (for instances). In this way, it has both error checking value of its own and serves as a preparation for the next step of semantic processing.

In terms of validation based on constraints, type checking makes use of local or static constraints to detect errors. For example, when a unary function for an object class is defined as "constant," if the compiler finds that there is an event that actually changes the function then it will complain. Conversely, it will complain if something is declared as "var" but there are no events to change it. This is static because it is based on the specification text rather than on the simulation traces which are dynamically generated.

In the following, we will concentrate on parsing and type checking. Though there are other things done at compilation time, they will be mentioned later at proper places, because they are more relevant to particular needs for, say, trace analysis or loop summarization.

12.2 Parsing

The DAO system uses an ATN parser, that is, a recursive descent parser plus actions associated with the parsing of certain constructs. The parser is implemented as a set of LISP functions.

The parser functions are generated automatically from a concrete syntax written in the meta syntax (described in the chapter on the DAO language).

The parser, in following a particular production rule, will not backtrack to every past decision point but only to the last one in that rule. If that fails it backtracks to the syntactic node at next higher level, which means the trial for this node is given up. The advantage of this is that it never need to undo anything. The shortcoming of not backtracking to all past decision points can be overcome by writing an appropriate grammar. In particular, by putting a proper keyword at a decision point and taking other measures, we can virtually eliminate all the need of backtracking within a production rule.

The parser produces an abstract parse tree in the form of LOOPS objects and instance variables (IV) of LOOPS objects. The original text is kept as an IV value of the LOOPS object for the node.

During the parsing, no type checking is done. For example, for a functional reference in the form of "x.v" where x is an object and v a property applicable to x, x.v is passed to the next step of processing intact. The type checking procedure will detect this when it can not find a declaration for the string "x.v". Then it will examine the string to see if it contains a ".". [1]

12.3 Type Compilation

One characteristic of specifications of real world systems is that they are "computation sparse" but "concept dense." One of its manifestations is that we have many user defined types for each specification. Here, by the term "type," we mean object classes, relation classes, and event classes. We chose to use the term type in the discussion of compilation because this makes it easy to draw analogies or make comparisions with traditional languages.

The large number of types warrants a separate stage for collecting them. This is done in two passes. During the first pass, simple types are collected and the types of their generalizations are checked to see if they are already defined. During the second pass, composed types are collected. For example, if "people" is a type and it is defined as "setOf person" then its validity would depend on the validity of type "person." So the type for "people" is processed in the second pass.

Type information is kept in a global data base so that it can be quickly accessed. But it then can be accessed by any group being processed (cf. the chapter on DAO language). This

[1] In Interlisp, "." is not used as a break character. This is why we need to go through this trouble.

is theoretically inappropriate because the language, even at higher levels of organization such as groups and projects, is supposed to be still block-structured. We would have problems if some renaming becomes necessary when a group includes another group.

12.4 Type Checking and Resolving References

Type checking and reference resolution are always done at the same time. Here the term "reference" can mean both a reference to an object or a reference to an event or relation. Whenever a reference is resolved, a link is established from the reference to the definition or declaration. The links will be used in setting up actual parameters for the formal parameters in a signature.

Checking Object Types

The complication of object type checking is caused by allowing generalizations (also known as supers). The checking of a type often requires going up a supers chain for a match.

When the object is of a collection type, the complications are compounded. The usual thing to do is to follow the supers chain of the corresponding individual type, e.g., to know if an object is of type "SetOf X," we may try "SetOf XX" if XX is a super of X. However "SetOf XX" may not be named as such, it may be named as "FOO." The type checker has to be aware of this.

Checking Predicate Types

In a specification language aimed at describing real world domains, name overloading is almost inevitable. First, there is certainly a type difference between a person entering a room and a set of persons entering a room. On the other hand, under normal circumstances there would be no reason for us to reject the word "enter" for the two kinds of events. "Enter" and "EnterSet" would work in this case, but the same argument for using messages in object oriented programming would argue against it.

When there is name overloading, the type checking has to resolve the reference to the right definition or detect an error if there is a conflict caused by multiple inheritance (cf. the chapter on DAO language).

12.5 Type Usage Checking

Type usage checking is based on a very simple idea, that is, a defined type (class) or a declared instance should be used. It turns out that it is both unexpectedly useful and unexpectedly complex. As a matter of fact, for the expense we are willing to spend on usage checking we can only do it for types and in a static sense. We can not easily check the usage of functional references, not to say checking the usage of axioms.

12.6 Interactive Compilation

For efficiency considerations, the major part of compilation is better done in batch. However, the system allows minor modifications be done both interactively and efficiently.

Incremental Compilation

In particular, one can modify a definition of a predicate if its signature is not changed, since, in that case, the link from the references will still be there and valid. When the signature of a definition has been changed, the sytem will automatically remove the old types, and put in new types (We note that for each definition there is more than one type involved. For example, if the definition is for an object class. The user may have defined some properties and status for the object, as well). It should remove all old assertions, as well. Ideally the system should be able to automatically recompile all the definitions relevant, but this is not implemented yet. However, it does have a record, so that it can generate a warning.

Mini-Parse and Breaking Facility

Due to the large number of syntactic rules, it is not easy to pass even the parsing stage. The system has the following facilities to aid users.

Before the parser works on the complete text of a definition, there can be a mini-parse stage which only looks at some key places in the definitions and ignores any syntactic errors elsewhere. The mini parser will even create a node in the object hierachy, if a specified super class has not yet been defined.

After a specification has been mini-parsed, the user can browse through it in some interesting ways. Thus the mini-parse stage enables the user to think about the content first and worry about the exact syntax later.

One of the syntax debugging aids that the system provides is in the form of displaying the syntactic rules for the relevant definition in graph or list. Another and more powerful aid is to allow the user to inspect the partial parse tree and to break the parser functions at appropiate places. When reaching the break, the offending expression can be singled out for editing and parsing. When the expression gets parsed, it can be copied into the definition from a buffer. These features have been explained in detail in the chapter on the interactive environment of DAO.

Chapter 13

Specification Simulation and Analysis

13.1 Overview of Specification Simulation

13.1*1 Role of Simulation in the Semantic Processing of Specifications

To be able to apply domain constraint rules to a specification, we need to have a complete and unambiguous account of events that happened to the individuals of concern.

The specification per se cannot serve this purpose directly for two reasons: 1. If the precondition ("NecessaryCondition") is not met, a specified event may not actually happen; 2. Constructs such as concurrent events, quantified events and collective objects complicate the problem. For example, the same object may appear as an independent individual at one point, but as a member of a collection at another point. The component events in a concurrent event may form a certain number of provably possible orderings while other orderings are just impossible.

We introduce a relation-based *symbolic simulator* to solve this problem. The simulator tries to step through all the specified events and relations to make sure that they can be realized. The simulator then collects the possible sequences of events so that analysis based on domain constraints can be performed on them.

Besides meeting the need of analysis, simulation helps to discover errors and omissions in specifying system constraints and enhance a uniform level of abstraction. For example, if one specifies that to be able to seat customers a table cannot be "occupied," the relation instance of the table being "occupied" needs to be asserted somewhere or can be inferred from other facts. Also, if the table is specified as being clean, then some time during the restaurant operations the table should be cleaned. Otherwise, the writer has to be content with simply saying that a customer is seated at a table, *without any possibility of mentioning details*. Similarly, it checks completeness because if one fails to specify all facts necessary for an event then the event will not be enabled and all the remaining events cannot occur. The simulator can also detect errors of incorrect conceptualizations if these errors will result in situations where some specified events cannot occur (or are not enabled). For example, if one specifies that for the customer to have a menu, the menu should not be already in another person's hand, then the formulation

Vxrperson [-i[at(Menu, x)]]

will not work since presumably it will have to be in many waiters' hands first. The inability to prove the formula will show to the writer that the type of x should be "customer" instead. But we note that the way the simulator detects an error is similar to the compilation of ordinary programming languages: the error message tells the user that something is wrong but it is up to the user to identify the exact cause of the problem.

The reader is forewarned that both the simulation and the succeeding analysis are partial in the sense that the simulation can not try out all possible sequences of events and the analyzer can not make use all known domain constraints on those sequences that are simulated. This is because, in simulation, to exhaust all possible orderings of events is a computationally intractable task. For the analysis, because our language does not force all categorizations to become explicit we may not be able to use the constraint entailed by the categorizations. Sometimes we may use heuristics to infer the categories of entities being described but we do not always succeed in doing that. However, given our particular problem domain, our scheme is still useful. This will be explained in the forthcoming section comparing symbolic execution and symbolic simulation.

13.1.2 Basic Notion of Symbolic Simulation

In a relation-based simulation, a system is simulated by both attribute values and relations among objects. The simulator verifies the precondition for a primitive event before the state of the next moment is calculated (for values) or asserted (for relations). This verification may involve intensive theorem proving activity. Once the precondition for an event is verified, the resulting new system state will be simulated by properly asserting or unasserting facts into the knowledge base. Another task of the simulation is to find out the right order for a set of events specified as being concurrent. An ideal algorithm should be able to find all possible orderings and eliminate impossible ones. A practical algorithm should at least find a subset of possible orderings which are most relevant to our interest.

The above intuition is based on the fact that we represent the history of a system as a composed event. In a history that uses only temporal connectives "A.", "&", ";", **and** T, we can show that to prove the possibility of the occurrence of the whole history is **equivalent** to proving the occurrence of the individual component events in their specified **temporal** orders. For example,

Vx EX [e(x);ei(x)]

is equivalent **to**

[VxGXe(x)];[Vx€Xei(x)].

In general, for the operàtor ";'\ if we represent the collections of system state as R,[1] **then** the fact that R precedes an event composed by ";" **will follow**

[R h+[e ; ex] **]≡[R** H+e]A R(|e|) ^**ei].**

(The reader is reminded that R(|e|) is the class of relations that hold after a time interval |e| and the time is relative to the time point when R holds.)

This equivalence is not readily transferable to events connected with "&", i.e., concurrent events. This is because there are many possible permutations of event orderings in those component events. Among these permutations some can be eliminated because they cannot allow the whole concurrent event to be enabled. That is, if a particular permutation *w* has two parts 7Ti and fl*2,

71*1 ; 7T$_2$ ≡ 7T

then *n* would be a subclass of the specified concurrent event class E, and it is possible **that**

$\neg[\,\mathrm{R}(|\pi_1|) \mapsto \pi_2]$.

The next problem is that for all TTS that make E enabled, they may not all be acceptable to the later simulation. This is because the resulting system states would be different for different TTS even if E can be enabled for all of them. A later event may need the state resulting from a particular *ir* for it to be enabled.

Our actual handling of concurrent events, due to their extreme complexity, is simplified. We will use heuristics instead of going through all possible permutations.

13.1.3 A Comparison of Symbolic Simulation and Symbolic Execution

First, we point out that our simulation, as the system now stands, only does partial semantic processing. For example, only a very small subset of the permutations of component events in a concurrent event is actually tried. This is not a simple implementation problems. It is a fundamental limitation of the scheme that one just cannot try "all the possible permutations." In fact, the limitations are the same as those to symbolic execution of ordinary programs. For example, both face binary branching factors for conditionals and exponential

[1]It can be viewed as the abstraction of the actual situation It(t).

explosions for concurrent events and indefinite number of repetitions [Taylor 1983]. Symbolic execution technique cannot effectively handle the simplification of logic and arithmetic relations in conditionals [Clarke]. Given the sophisticated theorem proving facilities offered by AI this obstacle may be partially removed, but the theoretical difficulties remain anyway.

However, the difference in their application domains makes their roles in error checking substantially different.

In a real world domain, we care about the possibility of an event happening as well as its exact numerical result. For example, if the customer does succeed in acquiring a meal in a hypothesized (i.e., designed by our specification) restaurant, how much he pays and how long he stays etc., may be relevant. However, the possibility of the happening of an event is the first concern. Simulation helps in this way. On the other hand, a computation system seldom has a problem in continuing its operation. The only exceptions are "array index out of bounds," "divided by zero," etc., which is a very small percentage of semantic errors.

13.2 Simulation Algorithms for Various Language Constructs

Top Level Data Structure and Algorithm for the Simulation

Our presentation of the simulation will stay at the level of algorithms. Some of the implementation decisions are discussed in the chapter on internal representation. The report on implementation status can be found in the section on experience with DAO of Chapter 10.

To give a high level description of the simulation algorithm, we must first introduce some definitions.

The history tree of a system is an abstract parse tree (see Fig. 10.4) with its nodes linked as defined below. In Fig. 13.1, for readability, we use directed edges for sequentially connected events that are actually sibling nodes in the tree. This is called the "history graph" (cf. the chapter on the DAO system).

A composite event node in the history tree is linked to another tree, which is derived from the abstract parse tree of the event definition and the trees for all its component events. This linking ends at primitive events.

To handle concurrent events, each event in the concurrent event is expanded to primitive events through the links we have set up. At any given moment, there is a set of primitive events that might be concurrent. This set is called the "currentSet." For each event in this set, there may be other events in the history tree that are specified as sequentially following it. They form another set of possibly concurrent events. We call this the "followSet."

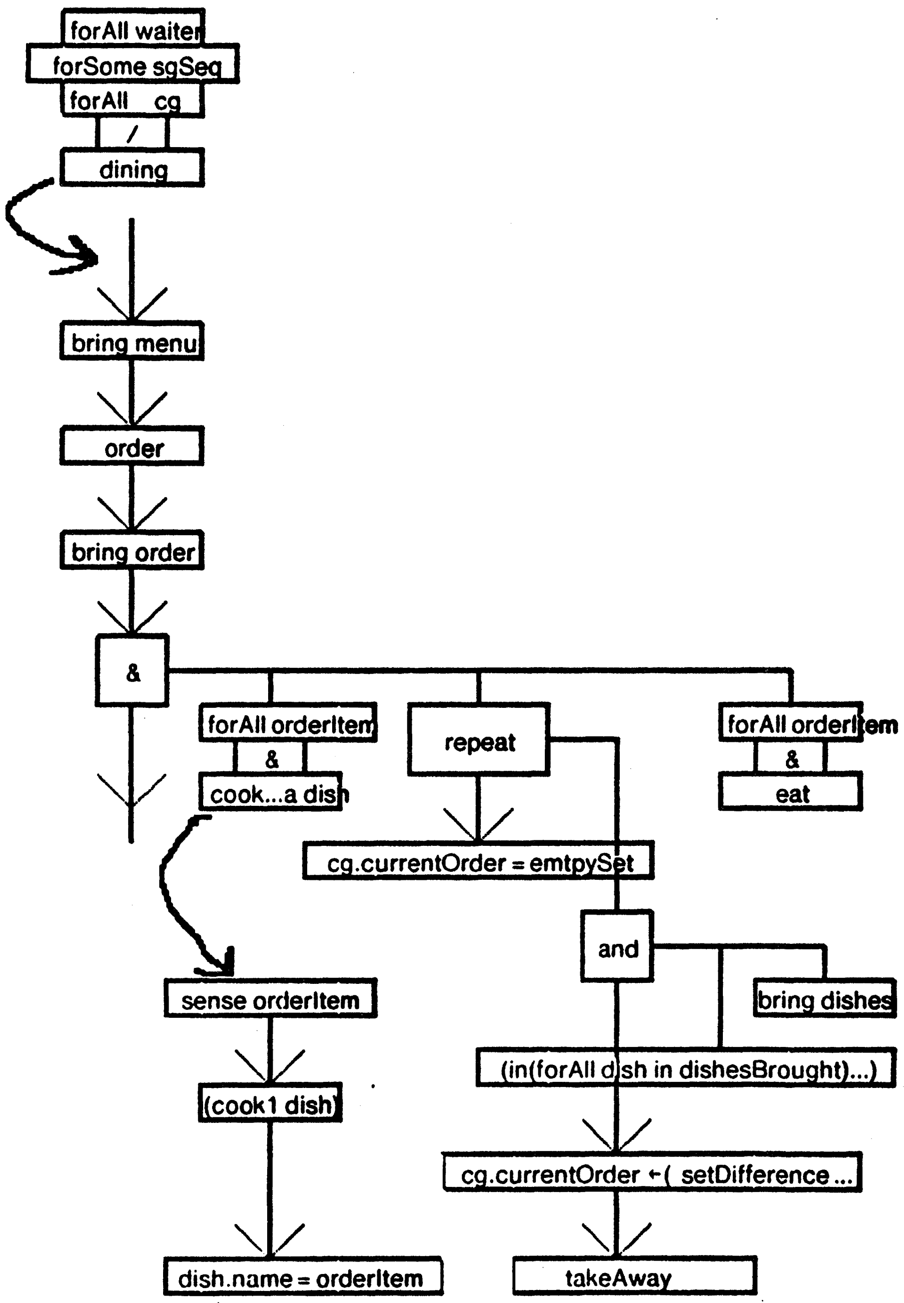


Figure 13.1: Part of The History Tree of Restaurant Specification

The followSet can be computed from the currentSet and the specified event orderings in the history tree. We note that some of the events in the currentSet might not actually be enabled.

When some events in the currentSet are found to be not enabled, they will be marked as "wrong order" and put into a set called "nextSet" (meaning the "next" set to try after the currentSet) and eliminated from currentSet. Once all the events in a currentSet are considered (i.e., simulated), the set will then be called "precedingSet."

Table 1 shows a pseudo-Pascal version of the top level procedure. Note that some parts contain recursive calls. To avoid complicating the algorithm, we do not consider conditional branches. A flow chart that handles conditionals is presented in Fig. 13.2

Primitive Events

The steps for simulating individual primitive events in the current set are:

1. Prove separately the preconditions for each of the events.

2. For an enabled event, check if the event is marked as "wrong ordering." If yes, an ordering between the event and an event in the precedingSet should be asserted.

3. Assert and then propagate the consequence. Consistency checking can be done here, also.

Concurrent Events

As we explained earlier, we cannot simulate concurrent events completely. Basically, we will try some of the possible orderings. If that succeeds we will proceed with simulating events that follow. The resulting system state may make future events unable to happen. In that case, we backtrack. To select the orderings of events to try first, we make use of (1) the temporal order predicates asserted as axioms, (2) the textual ordering of the events in the specification of a concurrent event. As an example of the latter, in the specification of a restaurant, the cooking, serving and eating may be specified as concurrent. Nonetheless, the textual order in which these events are enumerated suggests that for an individual dish this textual order is the actual order.

Recalling the last section, the processing of a top level concurrent event amounts to the processing of progressive sets of events, i.e., "currentSet." The selection of a set of currently

The Procedure of SIMULATE-NODE(path):

```
For each sequentially ordered node N in the path do
  if (the node N is sequential)
  then begin
      for all child nodes of the node N do SIMULATE-NODE(N)
      end
  else (* the node N must be concurrent)
      begin
        compute the first followSet of the node N ;
        initialize nextSet to empty set ;
        while (followSet is not empty) do
          begin
          currentSet := nextSet ∪ followSet ;
          for each node n in currentSet do
            begin
            if (the node n is not enabled) or
              (the node n disables other nodes in currentSet)
            then begin
                mark the node n as "wrong order" ;
                nextSet := {the node n} ∪ nextSet ;
                currentSet := currentSet − {the node n}
                end
             end ;
          for each node n in currentSet do simulate n;
            (* simulate the individual primitive events in currentSet)
          compute followSet for the node N ; (*from the currentSet)
          end;
        if (nextSet is not empty) then handling-error ;
      end;
```

Table 13.1: The Top Level Procedure for the Relation-based Simulator

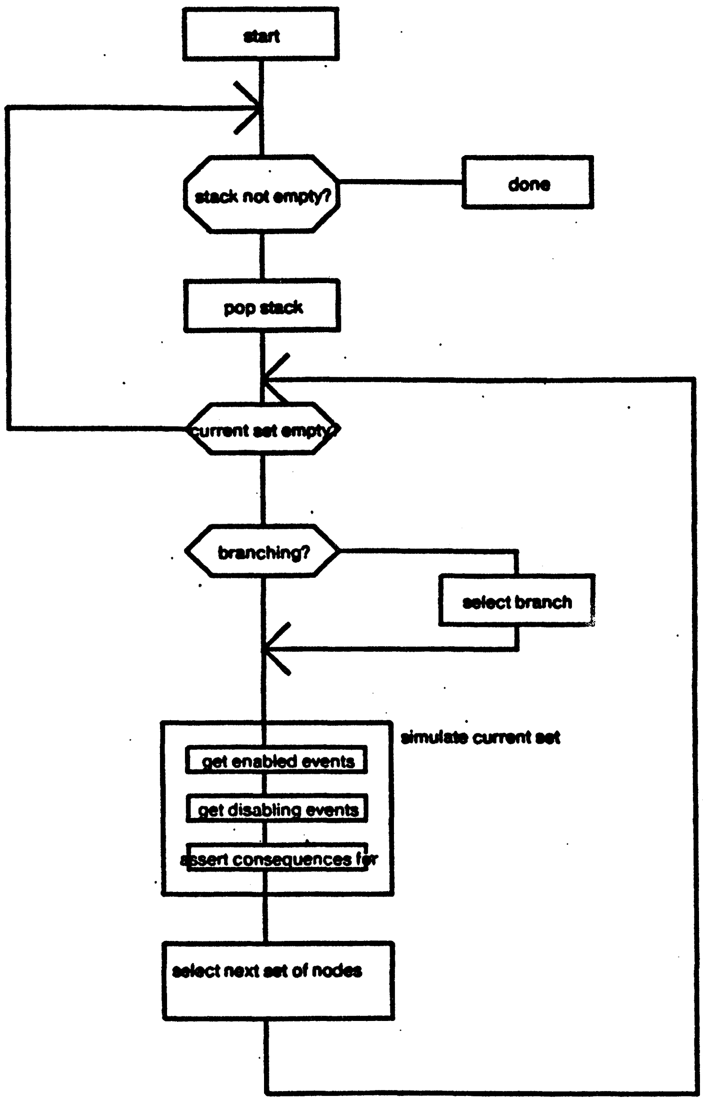


Figure 13.2: Top Level Routine of Simulation

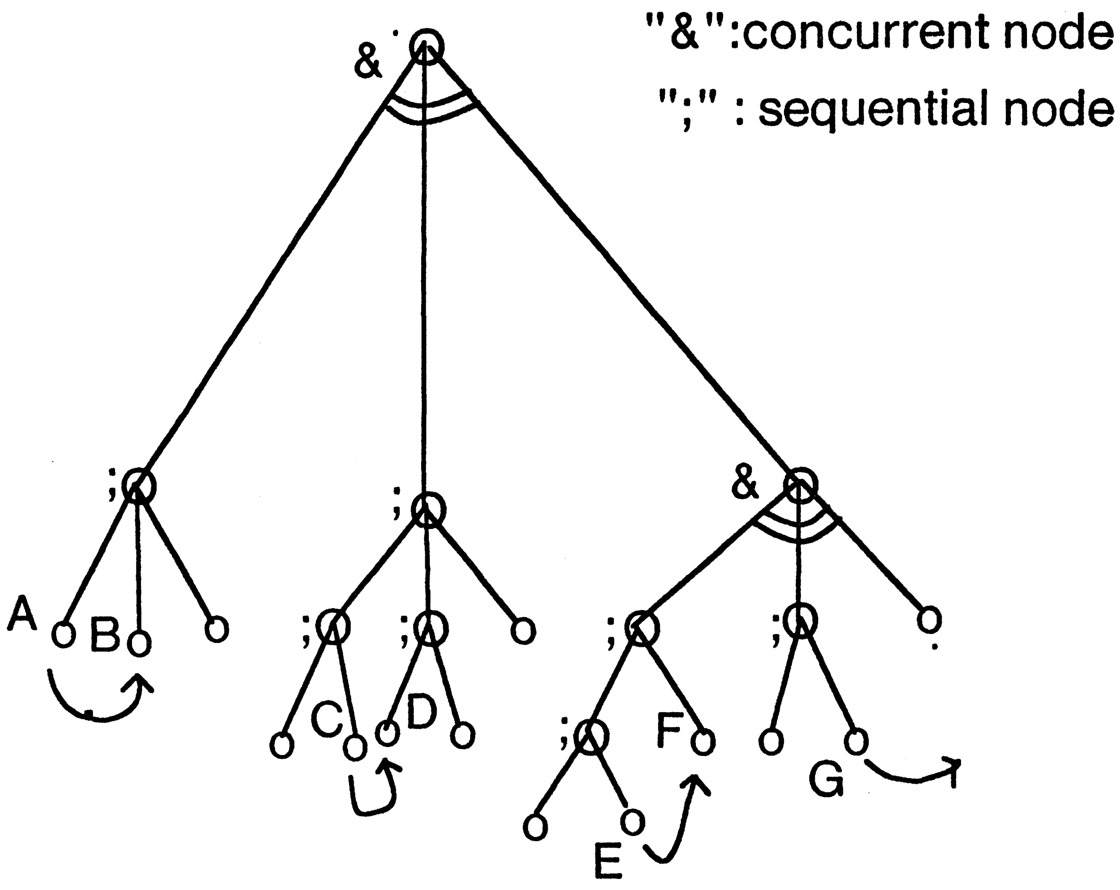


Figure 13.3: The Cases in Getting Following Set from Already Simulated Set

supposed concurrent (i.e., specified as "currently concurrent") events[2] is shown in Figure 13.3

This selection is basically a tree operation. There are three cases depending on what are the events that are just simulated.

1. The simulated node has a sibling that should be tried next. (From A node to B node in the figure.)

2. The simulated node has no sibling left but its ancestor within this concurrent event has a right sibling. The first leaf node of that sibling should be tried. This first leaf

[2]They may be the result of expansions of composite events.

node may be simply a child of the ancestor node (case of E to F). Alternatively, it might be a node several levels lower (case of C to D).

3. The simulated node is the last leaf node in the concurrent event at top level; no node is selected in this case (the case of G). (The case when the leaf node is in a concurrent event which itself is within the top level concurrent event is only a special case.)

The selection of the first set of events in a top level concurrent event is not difficult. It will not be shown here.

The checking of enabling of individual component events is just proving the preconditions for each of them.

The checking of disabling is complex. Theoretically, for each individual event, all possible orderings of the remainder events in the set should be tried. This is because the effects of two events taken together may falsify the precondition of a third event.

To do this checking, one has to set up a hypothetical context in the knowledge base. If the simulation results in unasserting facts in the original context (included by this hypothetical context), the processing becomes very complex and messy.

Because of this, the actual system only checks whether *one event* disables any other events. If there is a need of unasserting a fact, it is faked by asserting a negation of the fact in the hypothetical context. Because the theorem prover does a "disprove" first, there will not be contradictions during the checking. The hypothetical context will go away when the checking is done.

Repetitions

Semantically, both of the syntactic categories, "repetition"' and "whileLoop," are repetitions. We call the event inside a repeated event the *body* of the repetition. E.g., for event

"Repeat e $\equiv$ E,"

e is the body for E. Each happening of the repetition body is called a *round.* The relations asserted before a repetition are its initial condition and the relations right after its. termination, the final condition.

The algorithm is based on the following observations:

(a), if a part of the body of the repetition affects only distinct objects on each round, then it is equivalent to an event affecting the elements of a set, with the set being the collections of those distinct objects. The repetition can be viewed either as an event occurring to a collection object or a quantified event with the set as the domain of the quantification. (In

[Manna],the domain of a quantification is defined as the set of values that can be assumed by the quantified variable.)

(b). if, in a part of the body of the repetition, the same object is affected on each round of repetition, then the total consequence of the repetition might be computed (at "run time") by using the algorithm employed in ordinary symbolic execution [Clarke]. This usually involves setting up and solving recurrence equations.

This suggests that we should treat this kind of object or variable differently. We call variables affected in each round "loop variables." In the restaurant example in chapter 9, the object "dishesBrought" is a loop variable.

Theoretically, the algorithms should do the following things:

1. Check whether each round is possible. I.e., check (a). whether within one round the preceding events enable succeeding events, and (b). when it is the end of one round and the final condition is not met, whether the first events in the body are enabled again.

2. If the repetition does not have a definite number of rounds, but has a final condition instead, the condition should be proved for some indefinite number of rounds to ensure the termination of the repetition.

3. For each loop variable, the changes of both objects and attributes (which can be pure values) should be accumulated.

The actual processing can only handle some of these requirements. This will be discussed in the same order.

1. For (1.a), only the first round is checked according to normal procedures. The resulting final state (at the end of the round) is used to check the possibility of starting the second round. In this way, (1.b) can be partially checked.

   If it is a whileloop, since there will be an alternative path which does not contain any round and the resulting system state may be substantially different, we will introduce new contexts, i.e., treat it, in a way, as a conditional.[3]

2. To verify termination of a repetition, we observe that most of the termination conditions are predicates over loop variables. This is because that if they are not, the loops will either never be entered or never be exited. A comparison should be done

[3]We are aware that, in general, any while loop will introduce an indefinite number of branches.

between the values of loop variables required by the termination condition and the actual final values. The way of computing the values is discussed below.

3. Presumably the changes are recorded through "FunValChange" since any other **way** of change is not able to establish a relation between a previous value of a variable and its new value. If there are complex numerical relations among a large number of loop variables then the problem is a problem more appropriate for ordinary numerical programs. We would be unable to handle the problem successfully. If the changes are simple, a recurrence equation may be set up.

   For each variable, (1) if no termination condition is asserted then the accumulation is to infer its value at termination time. (2) If both initial and final conditions are known for the variable, and there are unknowns in the recurrence equations then solve the equations for the unknowns. (3) If all relevant relations are asserted, check the consistency.

   In the implementation, the computation of accumulations can be done at the end of simulating a repetition. Because a new assertion will often undo an old one, however, the information is gleaned on all the way of simulation. For each attribute value that changes (we know this from a static scanning), a list should be built for this purpose. The initial condition should be in the list and any assertion regarding this attribute should be put in. The assertion can be a LOOPS object, if it is at top level or a list if it is a consequence of an event. Presumably the terminal condition is at the end of the list. This list will be used independently of other lists of the same kind.

As rioted above, a strong assumption has to be made about repetitions. Namely, they have to be taken as primitives when appearing in a concurrent event so that there will be no interactions among objects inside the repetitions and those outside. Because this assumption is very strong, DAO will check for possible interactions and give warnings. It uses heuristics to eliminate possibly spurious error messages. For example, an agent's spatial relations are often not assumed to have a gap if it appears in both a repetition and another event in a concurrent event.

Conditional Events **and Disjunction of Events**

In a large composed concurrent event, for a set of events specified as possibly concurrent (i.e., the current set), each of the events is checked for being conditional. This is shown in figure 13.2. If it is one, a branch will be taken and the other branch will be pushed onto a "branching stack." Other information regarding the state of the simulation will be put on the stack, too. The simulation now takes one of two possible traces and continues. When the simulator is done with one trace, (i.e., when it cannot find the next node.), it goes back to pop the stack and simulates the other continuation of the trace. The process goes on until the branching stack is empty.

It would be desirable if some of the conditions could be collapsed by proving the equivalence or implication relations among them, but this has not been implemented.

Disjunctions of events also cause the traces to branch and are treated similarly.

Quantified Events

We have discussed extensively the proofs of preconditions of events with quantified variables. Because we can often use implicit quantification, a quantified formula is not too much more complicated than a formula containing constants[4].

We only mention several limitations.

In a quantified event, individual events often share common participants. For example, in Vx (&) e(A,x), presumably A will be involved in each event of type e. It is also possible for the arguments xi,X2,X3 to interact with one another. The actual checking system assumes that there are no interactions of any sort among elements in a collection. Moreover, the relevant state of the common participant, in this case A, should be declared by the user as "don't care."

For example, if e in the above is "cooking" and A is "Chef," we assume that the checking opeartion does not care how Chef is to cook all the dishes. This is because we may get inconsistent states of A if all the events are allowed to happen in an arbitrary way. Declaring "don't care" will save us the efforts to check them.

In this way, the processing scheme works as if it were dealing with an arbitrary member of the collection (which is the domain of quantification).

But since we are dealing with a collection anyway, this processing has essentially undergone some simplifications. For example,

(forAllxinX)(&)(Ex)

is processed exactly as

(forAllxinX)(Ex) or

(forAllxinX)(;)(Ex).

[1] Of course, at the price of treating all formulas as potentially quantified!

In the former case the system may do some additional checking to make sure things can happen simultaneously for the participants of E, but that is the only difference.

Lastly, the proof for concurrent quantified events (e.g., the case of Chef's cooking, Waiter's bringing and Customer's eating) will not produce a result that orders those quantified events. We will be reasoning about an arbitrary element of a collection and the result tells us the orders of individual courses of events that would happen to individuals.

In looking at the actual reasoning process this is not immediately obvious. There we assume all dishes are cooked, form a dishset, and the dishset includes the smaller dish sets brought to the customer each time. If we did not introduce the dish set, the inference would not be able to continue. But this is not to say that the bringing occurs after the whole dishset has been cooked.

This is acceptable because, here, we do not and *cannot* make any more specific claims about the macroscopic behavior.

Ordering Constraints in Composed Events

Ordering constraints in the form of axioms are used primarily for double checking the order of simulated events. When some order between events is found missing in a specification, the writer is to add it in these constraints. They are also used for guiding the ordering of events in disabling tests.

To check ordering constraints themselves we try to find a cycle in the graph formed by using $\prec$ and $\succ$ relations. A cycle means a contradiction, since all the events are to form a partial ordering.

Relations and Conjunctions of Events

In a composed event, in particular, a history (which is the target of our simulation), some assertions are relations. Assertions of this kind are indispensable. Some relations are true for a class of events, so we can put them into the consequence of the event, as part of the definition. But some relations are true only for a particular event. In that case, "E ! P" is a handy expression. In this formulation, P is true right after E. If we are to mean the meals cooked by the chef are accessible to the waiter, we should use "!" because the meals do not exist at the time cooking starts. On the other hand, if we say that the waiter brings a set of dishes to the customer and this set is a subset of, say, dishesCooked, then the formulation should be "E $\underline{\wedge}$ P" because the relation holds the moment that the event starts.

We note that conjunction of events carries the temporal implication of being simultane-

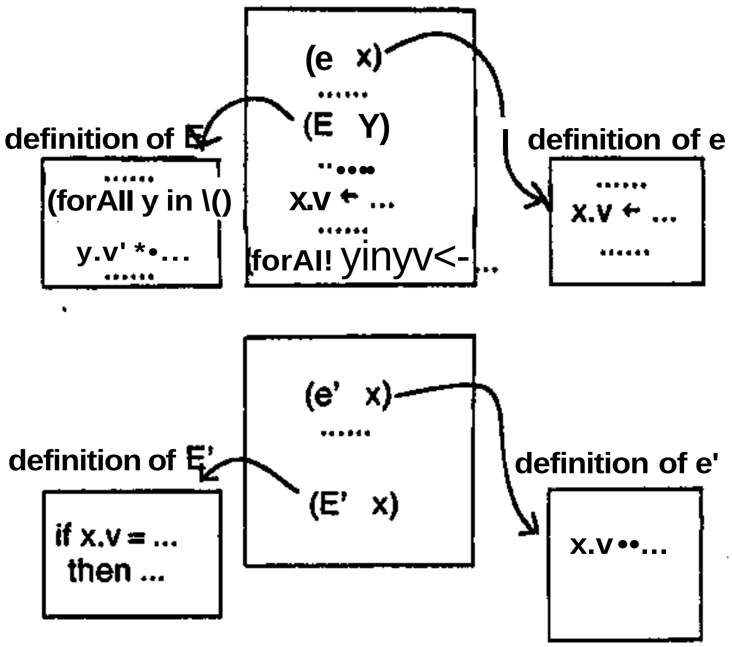


Figure 13.4: The Definition and Use of Functional Reference

ous, therefore, the events can not disable or enable one another. However, they are subject to other constraints that possibly restrict the same objects from participating in different events.

Resolving Functional References in Simulation

As shown in Figure 13.4, functional references can be defined and referenced almost anywhere in a specification. In particular, the definition of a value may appear in a lower level construct (a linked definition) or in the main text. The use of the reference can be in another linked definition or in the main text.

The scope of a functional reference is the scope containing the function definition. In this way, it is the only dynamically scoped category in our language.

In our implementation, a list is kept at the top level for resolving functional references. However, because the arguments of functions are all objects, rather than variable names, there is no identity problem.

13.3 An Example of a Simulation

As an example, we simulate the concurrent event in which some dishes are cooked, brought to and taken away from the customer, and eaten. The specification text is in Section 1 of Chapter 9.

1. **Step 1.**

   currentSet = {cook, bring, eat};

   No dish exists yet, so "bring" and "eat" cannot be enabled. They are put in the nextSet. The event "cook" is enabled, i.e., an arbitrary member of the dishesCooked is cooked and is in the kitchen.

   enabled events = {cook}.

   unenabled events = {bring, eat}.

2. **Step 2.**

   currentSet = {accessible/dishesCooked, isSubset/dishesBrought, bring, eat}.

   The two relations should be simulated unconditionally before proving preconditions for "bring."[5]

   The event "eat" is not enabled, since no dish is at the table.

   The precondition for "bring,"

   (accessible dishesBrought waiter),

   is enabled because it is true that

   (accessible dishesCooked waiter)

   and because of the rule relating a set and its subset:

   (if (and(at $S $p)(isSubset $s $S)) then (at $s $ p)).

   enabled events: {bring}.

3. **Step 3 Summarizing.**

   A demon "summarizing" is associated with value changes in repetitions. It is called to calculate accumulations for loop variables.

   For the loop variable, customer's "currentOrder," its initial value is the content of the object order and its final value is ∅.

[5]The notation such as "accessible/dishesCooked" is used in making a list into an atom so that it is easier to process in Lisp.

We know from the first quantified event that

  {name(d) | d∈dishesCooked}= order.[6]

In actual implementation it is an assertion

  (ontofun nameInverse order dishesCooked),

meaning that the set of order items (in order) is the result of the onto mapping "name" from dishesCooked.

We set up a recurrence equation:

  Δ cg.CurrentOrder = (name dishesBrought).[7]

Determining the unknown in this equation, we get

  (Union $dishesBrought) = dishesCooked

However, dishesTakenAway cannot be summarized in this way.

4. **Step 4.**

  currentSet = {eat, takeAway}.

The precondition for event "eat":

  (at $dish@dishescooked cg)

can be proved now.

  enabled event = {eat}.

We do not even know the identity of dishesTakenAway. We can either ask the user or, in this case, we can try to find a relation between dishesTakenAway and some object with known identity. We actually infer that

  dishesBrought ⊂dishesTakenAway

from the ordering constraints

((forAll x: dish)((bring dish) < (takeAway dish))).

Therefore "takeAway" can be proven enabled. However, it will disable the event "eat."

The problem is more serious than disabling. Because either event nullifies the object involved (the dish), and no event can ever happen to the object anymore. We now have found an error in the specification which should be corrected by the user.

[6]As we have pointed out many times, the object order is the counterpart of a physical form and has its identity. It is through some twist that we treat it as a pure value here.

[7]For the ease of processing, we allow a function take a collection as its argument if the type of the elements of the collection is an acceptable type. Thus we run the risk of mistaking definitions if there is a name overloading, of course. This kind of "dirty code" is only allowed in several places.

The interactive environment may stop to ask the user what to do. The specifier may want to add a state "empty" for a dish and redefine "eat" as an event that changes the dish (making the state "empty" to be true) rather than an event that makes it nullified.

The event "eat" is selected to simulate. The event "takeAway" is put into nextSet.
disabling = {takeAway }

5. **Step 5.**
currentSet = {takeAway}

Finally, "takeAway" is simulated.

13.4 The Analysis of Specifications

13.4.1 The Role of Trace Analysis in Specification Validation

As defined earlier, a trace is a dynamic account of a system's behavior. It is a complete and serialized instance of the system history.

The traces of system history are the target of validations based on domain constraints, especially global constraints.[8]

In a sense, if the simulation does checking based on what the specifier has said to be true the analysis would base its checking on what should be true in the domain or what the specifier has implied to be true. E.g., a specification may define an event "bring" that does not have the precondition that the agent is able to access the object (the ability to access can simply be that he is at the same place). Thus, a history allowing someone outside the restaurant to bring dishes from the kitchen to the dinner tables may pass the simulation because the inference can go through. However, spatial constraint checking may reject this at the analysis stage.

As in simulation, the processing here is partial. Moreover, only those traces that the simulation has tried will be available for analysis. However, trace analysis can work on a subset of traces and provide help in error checking. In this sense, as long as a dynamic account of system behavior can be generated, a simulator can do the preprocessing for trace analysis. A powerful simulator is helpful but not necessary.

[8]By "global constraints," we mean those cases where a relation or event is restricted or constrained by another one that is spatially or temporally separate from it. For example, the constraint that a sensing event should be followed by some action by the sensing agent is a global constraint. On the other hand, the type requirement for the arguments of an event is a local constraint. A local constraint can usually be checked statically.

13.4.2 Analysis Algorithms

Algorithms in Preliminary Processing

The analysis of traces depends on the knowledge of the categorizations of the objects, events, etc., in a specific system. This kind of information, e.g., the types of primitive events, being component or noncomponent for each object, etc., can be collected once and for all at compilation time. (As a matter of fact, some local constraints can be even checked at compilation time.)

Determining Types of Primitive Events

The type of a primitive event is determined by its operational consequence, which is reflected in the consequence part of the definition, but can also find its way into the event signature in the case of creation or disappearance events. The determination procedure goes as follows:

1. Check the consequence of the event.

   Ignore abstract relations. (Some relations are known to be abstract, for example, ">" and other mathematical relations.)

   If a relation is a composed relation, then expand it and check each component relation.

   If a relation is user-defined, ask about its relation type.

   If there is more than one type of relations, determine the type according to the following dominance order: external, internal, spatial.

2. Check the signature of the event.

   An event that is a creation or nullification (recognized from its signature) is at least an internal state change event.

3. An event definition may have failed to specify its consequence if no consequence is found for it through the above procedure.

In the actual compilation, we will also check whether an event is primitive so that we would not make a mistake in determining the type. If the signature of an event has 3 or more arguments, *and* one of the arguments is an information object or the event is a creation event, then it is very possible that the event is not a primitive event.

Trace Generation

When there are multiple contexts involved in the simulation, the events and relations in **a** trace are collected by going from the starting context to the ending context and taking **a** particular combination of branches each time.

To generate a trace, for each fact in the trace, all the forms representing the same identity have to be converted into a common form. In the formulas no quantified variables are allowed. For example, if we know that the union of all the sets generated as "dishes-Brought" in the repetition of the restaurant example is identified as dishesCooked, then the event "bring(waiter, eg, (union $dishesBrought))" should be changed into bring(waiter, eg, dishesCooked).[9]

We note that, at the stage of analysis, in the current implementation, we do not distinguish a type from its corresponding collection type. E.g., we can have eat(eg, dishesCooked). This is to simplify the processing.

Algorithms **for Constraint Rule Applications**

The constraint axioms often involve both individuals and events. Their forms are often complex second order formulas. Also, even if we can view a trace as a predicate, to prove properties on predicates is often better done in a procedural way. Because of this, all the rule applications are put in procedures. Every rule has a procedure. The procedure is called for each relevant object for each trace. There can be efficient ways to combine the checkings and to make fewer number of passes, but it is not done in the current implementation.

We explain the actual constraint applications by three examples.

1. Noncomponent Creations and Nullifications

   This is one of the simplest cases. Our checking is to make sure that there is exactly one creation event preceding exactly one disappearance event for the noncomponent. Any event happening to the object should be within this time interval.

2. Component State Restoration

   To do this checking, the analyzer has to know which are the changed component states and which arc the new component states in the trace. This requires a search through

[9]If the union of dishesBrought (dishes that arc brought in) is not so identified, e.g., **if** we only **know** the union of dishesBrought is a subset of dishesCooked, we will usually have two sets to consider. **One** is dishesBrought, another is the difference set, i.e., (dishesCooked — dishesBrought).

all the facts derivable from the trace, because the alteration of the initial state of a system might occur late in its history.

When the system is dynamically stable, the problem becomes even more complex because the final or initial component states can then contain noncomponents.

3. Goal Satisfaction and No Unnecessary Path

This comes from the constraint that a system behaves purposefully.

To check that every goal is satisfied for each trace, it suflSces to prove the goals from the trace. In the case of goals being events, it is straightforward. In the case of goals being relations, a data base storing old facts (the facts that are not currently true) has to be kept. Another alternative is to try the proof at each simulation step and mark the successful ones.

To check the requirement the other way around, i.e., that each path contributes to goal satisfaction, one need first collect independent paths. The collection algorithm basically follows the definition of independent paths.

For any independent path, check if there is an event or relation that restores component states or satisfies goals. A path that has neither is an unnecessary path. It is important to note that although this checking will report an error as "unnecessary path," the actual problem may be that some causal effect of this path on other paths is not specified.

Chapter 14

Related Works in Specification Research

Many fields of computer science are closely related to the construction and analysis of system specifications. It is impossible to mention all of the specific works in this limited space. In the following, we will try compare our work with those that have been most influential in the relevant research directions.

System Specification Languages

The language DAO, purported to describe real world systems, belongs to the family of system specification languages. Among these languages DAO is unique in incorporating formalized domain constraints in its semantics and forcing a suprastructure on specifications. But it has a lot in common with other languages in its design and implementation.

The language most closely related to DAO is Aleph [Winograd 1984]. The structure of the Aleph syntax, the central place of events in Aleph, and the categorization of specified entities as objects, activities and relations in Aleph had a strong influences on the design of DAO. In this regard, one may view DAO as a variation of Aleph. But DAO does not share the same model with Aleph and this has an effect on the language design. For example, in Aleph, an observation of behavior can be abstracted to any level. We can have an activity "isSent" for a message. For DAO, for the observed event to that message, more than one object is involved, and at DAO's base level they all have to be present. Therefore "isSent" can only be the abstraction of the event sending of an object by another object (presumably an agent). Another difference is that Aleph puts more emphasis on the building of the specification environment, mainly the management of specification texts and parse trees, while DAO stresses various semantic checking.

Gist [Balzer and Goldman] and Delta [Holbaek-Hanssen Handlykken and Nygaard] are two other languages in the same direction. Delta has pioneered the description of real world systems. It is very specific in analyzing roles of system components. Gist has a rich repertoire of language constructs for expressing behavior patterns. In terms of processing, Gist has a behavior explainer and a simulator doing symbolic execution [Cohen] [Swartout]. Our language differs from these two in being more declarative. As a matter of fact, Delta looks very much like a normal programming language. Many constructs in our language are directly related to first order logic.[1] In terms of expressiveness, each has its own merits. To some extent, DAO can handle collection objects and concurrent events which add considerable complexity to the processing and even the definition of the language. The other two languages do not handle them. In terms of simulation, the Gist simulator works on what we called relations alone while our simulator works on events and a knowledge base containing time-dependent relations.

Software Specification Languages

Research in software specification languages, in particular, algebraic languages, has contributed mathematical rigor and elegance to specification language research in general. Many problems at intuitive levels are reformulated as algebraic or logic problems and are given lucid explanations. For example, the principles of initial or final algebras make it clear how object identities can be decided in building models. [Burstal and Goguen]

There are other languages such as COSY [Lauer, Torrigiani, and Shields] and GEM [Lansky and Owicki], which have been developed for describing complex system behaviors and are event-based. They often include some temporal models in their semantics.

In both cases, there are two deep gaps between our approach and theirs. First, we are interested in specifying real world systems where not only the input and output, but also the process itself, matters. On the other hand, for them, in principle, input and output (or argument values and function values) are what is there to specify. Secondly, they tend to use restricted forms to ensure the ease of manipulation and fine mathematical properties of specifications. This makes their specification not easily understandable. For us, understandability is one of the primary concerns.

[1]But some caveats have been discussed in the language definitions.

Knowledge Representation Languages

The work on knowledge representation languages provided most of the conceptual nutrients that nurtured the growth of DAO [Brachman][Bobrow and Winograd]. The difference here is that DAO is inclined to have a relatively narrower domain and certainly a much more restricted model. DAO intends to accomplish a well defined small set of tasks, namely describing operational systems and checking the descriptions for errors. Its semantic bases are narrow and formalized. Another difference is that knowledge representation languages are programming languages. They have rich control features while DAO does not have any.

Object-Oriented Programming Languages

As we have discussed in the introduction, although we share "terms" such as "objects" with object-oriented languages, "objects" are only entities in the computer for them while for us they are real things.

Environment

DAO owes as much to Aleph in the design of the environment as in the design of the language. But it seems both learned much from LOOPS, which is what they are built on top of. For example, the features such as browsing class hierarchy and providing "summaries" (cf. the chapter on environment) are present in LOOPS. As its basic interface design philosophy, DAO adopted that of interlisp-D [Xerox]. For example it uses pop up menus instead of pull down menus.

DAO differs from these environments in building tools for serving the need for simulations and theorem proving. In that way, the interactive operations of DAO are richer in content. For example, it allows the user to examine the contents of different contexts (cf. the chapter on internal representations). DAO also draws an event flow graph that is not a tree or a lattice structure.

DAO tries to index the definitions and their uses so that modifications can be guaranteed to be noticed and propagated. But this indexing is not as generally done as in Smalltalk [Goldberg and Robson] environment.

Simulation and Analysis of Specifications

The idea of simulation of descriptions is old. [Rieger and Grinberg] was the first to relate it to causality and discuss some primitives. But their simulation was basically running procedures that were attached to the components being simulated.

Its related idea in software engineering is making specifications operational [Zave]. By "operational," we mean that each step enables the next step either in a causal or in a rational sense. We believe that this is essentially the same idea as simulation. However, operational specifications are for software and are sometimes executable programs themselves.

Another form of this approach is planning in AI. As a matter of fact, a simulation can be seen as a plan execution [Fikes and Nilsson], and finding appropriate ordering for concurrent events is like achieving conflicting goals [Sacerdoti]. While the operators in STRIPS are similar to our primitive events in being characterized by preconditions and consequenes, the "plan" we simulate may contain collection objects and complex temporal patterns. In this "plan," objects at many abstraction levels may be involved, which is also not a feature of usual planning systems.

In the algorithmic aspects, DAO is related to three subfields of research: symbolic execution, concurrency control and program flow analysis [Taylor 1983][Clarke].

We have discussed the commonalities and differences between symbolic execution and our symbolic simulation in the chapter on simulation. The case with concurrency control and our handling of concurrent events is very similar. Briefly stated, although both are subject to exponential explosion in terms of a complete exploration of paths, symbolic simulation can gather helpful information even if it only partially explores possible traces.

Both in DAO and in program flow analysis, we examine whether an entity is created or defined before something happens to it. Typically, in program analysis, we require that a variable be declared before it is used, and its value be defined before it is referenced. In our discussion on causation, we have called this the "genetic principle."

But in DAO, geneticity checking is possible only because we have a conceptual scheme to put things correctly into specifications. What is hard is to get things right conceptually. On the other hand, in data flow analysis the complexity of finding anomalies using this principle is because of the algorithmic structure of the computation.

For example, in a real world system objects are organized at different levels of abstractions and the abstraction relations can be part vs. whole, symbol vs. interpretation vs. realization, instance vs. class and so on. Unless these relations are handled cleanly, there is no way to use the geneticity check correctly.

Suppose we are in the restaurant again; one may understand the statement "the customer orders a dish and the chef cooks the dish" as indicating *generation* of a new object "a dish." In that case we will have a perfect-looking specification from the data flow analysis viewpoint

```
(order customer (d: dish));(cook Chef d ),
```

since d is indeed first "declared" then "used" and checking "data flow" is the only tool that that paradigm has.

In our paradigm, problems such as this one are very unlikely to occur, in the first place. This is because one is forced to choose between information objects and concrete objects when one is defining any object class. Consequently one is unlikely to think that the event "order" involves the actual object "dish." Even if one makes this mistake, the spatial gap will force one to give an account of why a dish already existing should go to the chef first (this may take the form "(bring customer dish chef)" in the real specification) and so on.

Furthermore, geneticity checking is only one of many checking rules and this rule is applicable only for part of the system. E.g., a system component need not be generated in the history.[2]

What we have argued above is that the data flow analysis viewpoint cannot cover our general genetic principles. Nonetheless, it is an instance of it, and an interesting one. In fact, programming variables are one of the attributes of physical objects cells, in our operational model of computers. And attributes have values. These attributes, as abstract objects, follow the geneticity principle. On the other hand, the corresponding physical objects also follow the principle. In programming terms, a variable has to be declared and then be used (definition of a value or reference to the value).

[2]**Yes, it is defined in the specifications, but so is a non-component.**

Chapter 15

Conclusion

Two problems motivated this research. The practical problem has been to validate formal accounts of our intuitive ideas about real world systems. The theoretical problem has been the search for the semantics of artificial languages that do not run.

At first glance, this soil looked barren. There seemed to be very little that can be said about properties that are true of systems in general. And it seemed that there is little one can do in terms of mechanical validation once we are outside the realm of formal structures.

The results reported here are nothing sensational. The door has just been opened. However, we have shown in some modest examples that there is something to be found and that something can be done.

We have shown that formal descriptions of system behavior can be analyzed by exploiting domain constraints relevant to operational systems. The domain of operational systems covers a broad class of systems, among which are many physical systems, computer systems and office systems. Our representation and methods may be used whenever programs are designed to do reasoning about such systems.

Our framework is built on the notions of causation and interaction. We suggested using two levels of models to characterize the semantic bases of representation languages intended for real world domains. Namely, there is a mapping procedure from the observational model to the abstractional model. Causation is the single most important criterion in selecting what to map. We formalized various concepts relating to human activities and stable operational systems. These concepts, though simple, have given us a clearer understanding of the domain. Moreover, we derive some inherent constraints useful for checking the correctness of specifications.

We have built a substantial system to experiment with our ideas. Currently, running on a Xerox 1132, it provides a complete environment for a user to create, browse, edit, parse,

type-check, simulate and analyze specifications. The simulation uses a theorem prover to check whether specified events can happen according to the axioms specified by the user whereas the analysis component uses constraints inherent in the domain to check whether the axioms themselves violate domain constraints.

Some small examples have been analyzed and checked for errors. They are still not large enough for us to claim that our theory as a whole has direct applications to larger problems. However, based on the complexities our system is able to deal with, e.g., collection objects, multi-levels of abstractions of objects, the processing of these examples illustrates some basic principles.

This research was not intended to be directly transported to problems of real sizes. However, some of its simple heuristics, e.g., "no events can happen to non-existing objects," and some of its principles, e.g., "only put directly involved objects into the argument list of an event" can be fairly readily useful.

In addition, we see this work as forming a basis for system design that will apply throughout the application areas for computers, since, like all other systems, computer systems are based on physical causation. As some other authors (e.g., [Holt 1985]) have claimed, issues that at first sight might seem "too implementational" (such as physical location, and the creation, use and nullification of objects) are in fact critical to the design of understandable systems.

This research is just a beginning in the quest for the nature of systems from the perspective of computational analysis and experimentation. Because we did not have time to explore larger and more concrete examples, many conclusions are still shallow. Future research should concentrate on systems in one or two specific domains, e.g., business or biology, and see how the general constraints manifest themselves. It is hoped that new constraints will be discovered, as well. As the framework now stands, we can place it on top of normal knowledge representation systems, adding more structure to the representation of and reasoning about real world systems. This may be the practical direction to go.

Finally, we note that there are limitations to our methods. Technically, simulation algorithms cannot handle general cases of repetition and concurrent events and so on. In these cases, one still must test the system on actual data to be assured of its proper behavior on all occasions.

There is a fundamental limitation, as well. Our checking is based on heuristics derived from domain constraints, but not all errors are in violation of these constraints. Actually many of them violate only system specific constraints. In that case the only solution is to consult the intent of the user directly. But this is inevitable.

Appendix A

Conventions of Symbol Uses

A.1 Logic Symbols

$\wedge$ -- Logic AND (for its extended meaning in DAO, see $\underline{\wedge}$).

$\vee$ -- Logic OR.

$\oplus$ -- Exclusive OR.

$\neg$ -- Negation, or logic NOT.

$\equiv$ -- Logic equivalence.

$\not\equiv$ -- Logic non-equivalence.

$\rightarrow$ -- Logic implication.

$\forall$ -- Universal Quantifier.

$\exists$ -- Existential Quantifier.

$\Box$ -- Always.

$\Diamond$ -- Sometimes.

A.2 Set Theoretic Symbols

$\emptyset$ -- Empty set.

$\subseteq$ -- Set inclusion or subset.

$\subset$ -- Proper subset.

$\supset$ -- Super set.

$\in$ -- Set membership.

$\cup$ -- Set union.

$\cap$ -- Set intersection.

A.3 Special Symbols Defined in DAO

Operators

$\Rightarrow$ -- Cause.

$\mapsto$ -- Precondition.

$\Leftrightarrow$ -- If and only if ... then
(defined in the chapter on time, rarely used)

$\Leftarrow$ -- Only if ...then
(defined in the chapter on time, rarely used)

$\#$ -- Past.

$\underline{\wedge}$ (or simply $\wedge$) -- Simultaneously.

$\underline{\&}$ (or simply &) -- Concurrently.

$\underline{!}$ (or simply !) -- Immediately follows.

$\underline{;}$ (or simply ;) -- Follows.

: -- Contains. (in terms of the durations of events)

/ -- Overlaps. (in terms of the durations of events)

:; -- Starts earlier than. (but not knowing the times of endings of the events)

$\preceq$ -- Precedes or simultaneously.

$\prec$ -- Precedes.

$\succ$ -- Succeeds.

$\prec_!$ -- Immediately precedes.

$\succ_!$ -- Immediately succeeds.

:: -- Contains. (in terms of the durations of events)

// -- Overlaps. (in terms of the durations of events)

$\prod$ -- Product (of operators).

$|\ |$ -- Norm of ...

$\|\ \|$ -- Corresponding sequence of ...

Symbols for Different Types of Objects

θ -- Information object.

ϕ -- Physical object.

ψ -- Abstract object.

ω -- Object.

θ_ϕ -- Physical form of an information object.

θ_ψ -- Abstract form of an information object.

Symbols for Different Types of Variables

a, a_1, a_2,...a_i,...b, b_1, b_2,...b_i,...c, c_1,... -- Constants.

t, t_1, t_2,...t_i,...t_k,...τ, τ_1, τ_2,...τ_i,... -- Time variables.

Υ -- Set of time variables.

l, l_1, l_2,...l_i,... -- Spatiotemporal locations.

bool, $bool_1$, $bool_2$,...$bool_i$ -- Truth value indicators.

E, E_1, E_2,...E_i,...e, e_1, e_2,...e_i,...e_n... -- Events (classes or instances).

f, f_1,...g, g_1,...h, h_1,... -- Functions.

p, p_1, p_2,... p_i,... $p_{1,1}$, $p_{1,2}$,... $p_{1,k}$,...$p_{1,i}$...q, q_1, q_2,...q_i...-- Propositions.

u,... u_i, x, x_1, x_2,... x_i,... x_n,...y, y_1, y_2,...y_n... -- Object variables.

Appendix B

Formal Properties of Temporal Operators

B.1 Properties of the Connectives and Operators

The following connectives satisfy commutativity:

$\wedge$, $\vee$, $\equiv$, $\oplus$,

but these do not:

$\Leftrightarrow$, $\Leftarrow$, :, /, ;, !.

In a word, all the temporal connectives *for predicates* are not transitive.

The following connectives satisfy associativity :

$\wedge$, $\vee$, $\oplus$, ;, !, :, &, when.

Some of these are proved below, e.g.,

X;[Y;Z] $\equiv$[X;Y];Z

proof:

LHS: X;[Y;Z] (t) $\rightarrow$

$(\exists t', t_1) X(t) \wedge Y(t') \wedge t'-t \geq |X(t)|$ and

$Z(t_1) \wedge t_1-t \geq |[X;Y](t)|$

RHS: [X;Y];Z (t) $\rightarrow$

$(\exists t', t_1) X(t) \wedge Y(t') \wedge t'-t \geq |X(t)|$ and

$Z(t_1) \wedge t_1-t' \geq |Y(t')|$

Since

$Y(t') \wedge X(t) \wedge t'-t \geq |X(t)| \wedge t_1-t' \geq |Y(t)|$

by definition

RHS $\rightarrow$LHS.

Similarly LHS -+RHS. For &,

X&[Y&Z] ≡[X&Y]&Z

X&[Y&Z] (t) ≡X(t) A (3t')[Y&Z] (f)

≡X(t) A (3t')[Y(f) A (3t»)Z(t»)]

≡X(t) A (3t\ t")[Y(t') A Z(t")]

≡[X&Y]&Z.

Similarly we can prove

X:[Y:Z] ≡[X:Y]:Z

X when(Y when Z] ≡[X when Y jwhen Z

Transitivity is satisfied by the connectives : -<!,-<,!,; and so on.

Some interesting relations can be proven for temporal ordering rules and composed predicates. For example,

X&Y A X::Y ->X:Y

proof:

LHS —X&Y A OX&Y ->X:Y

-X&Y A X&Y ->X:Y

-*X:Y

Similar properties hold for X/Y, X;Y and so on.

We now list the connectives in the order of precedence. This is just a notational convention we adopt, not a proper part of our theory.

["S A , V J [:], [/], [&] [;, !], [repeat], [when], [;<, -<,, ->], [•, 0, #].

To indicate priority we use parentheses as [...], or {...}or "begin" and "end".

B.2 The Properties of the Norms of Predicate Classes

When introducing "if then", we implicitly defined the norm of a primitive class or a class constructed by primitive classes and logical connectives. The norm is a function that maps a time point to a length of interval (represented as integer in our theory). The integer is the duration of the instance of the class.

A basic property of the norm for a predicate class X and time t is:

$\mathrm{min}_t|X(t)|<|X|(t)<\mathrm{max}_t|X(t)|$,

where min^ is the minimum of |X|(t) and max* is the maximum of |X|(t).

Let Max and Min be the functions that return the maximum and minimum of two values respectively, obviously,

|X A Y| = Max(|X|,|Y|).

It turns out we cannot further define norm for other compound classes in a consistent way. For example, $|\neg X|$ is just $|\neg X|$.
However, there are some boundaries:

$$\mathrm{Min}(|X|,|Y|) \le |X \vee Y| \le \mathrm{Max}(|X|,|Y|)$$
$$\mathrm{Min}(|X|,|Y|) \le |X \oplus Y| \le \mathrm{Max}(|X|,|Y|).$$

Many other properties hold, for example,

$$|X;Y| \ge |X|+|Y|$$
$$\mathrm{Max}(|X|,|Y|) \le |X/Y| \le |X|+|Y|$$
$$|X{:}Y| = |X|$$
$$|X\ !\ Y| = |X|+|Y|$$
$$|X \text{ when } Y| = |X| = |Y|$$
$$\mathrm{Min}(|X(u)|) \le |(\text{forSome } u)X(u)| < \mathrm{Max}(|X(u)|)$$

where u is an arbitrary member of U when free. Similarly,

$$\mathrm{Max}(|X(u)|) \le |(\forall u)(\&)\ X(u)| \le |(\forall u);\ X(u)| \le \Sigma|X(u)|.$$

Appendix C

The Concrete Syntax of the DAO Language

Production Rules for Regular Syntactic Classes

<ArgumentReference> =: (<<ATOM0>> @name)
<AtomReference> =: (<<ObjName>> @reference)
<BinaryBoolPredicate> =: (<<IsBinaryBoolConnective>> @connective)
        (<Predicate> @predicates) *
<CapacityDeclaration> =: (<NamedDeclaration> @capacity) ;
        <<Maximum>> : (<AtomReference> @maximum);
        <<minimum>> : (<AtomReference> @minimum)
<ChangeFunVal> =: [(<FunctionalReference> @function) |
        (<ObjFunctionalReference> @function) |
        (<AtomReference> @function)] ← (<Reference> @value)
<Character> =: (<<CHARACTERP>> @item)
<CollectionType> =: (<<CollectionTypeKEY>> @composition)
        [(<<ObjName>> @type |
        (<AlternativeType> @type)] {suchThat
        (<Predicate> @modifier) * }
<CompositeEvent> =: EventClass (<EventSignature> @signature)
        {(<ImplicitArg> @implicitarg) }<DEFINITION> <AXIOMS>
        {(<THEOREMS> @theorems) }
<CompositionType> =: (<<CompositionTypeKEY>> @composition)
        (<Declaration> @type) *

```
<ConditionalPredicate> =: if (<Predicate> @premise)
        then (<Predicate> @consequence)
        {else (<Predicate> @else) }
<ConstRef> = <Numbe> | <Integer> | <String>
<Declaration> = <Type> | <NamedDeclaration>
<Default> =: DefaultTo ([form |content | frame | physicalForm] @default)
<Definition> = <Inclusion> | <ObjInstance> | <Episode> | <History> |
        <Overview> | <RelClass> | <EventClass> | <ObjClass>
<Enumeration> =: Set (<<ElementName>> @element) *
<Episode> =: Episode [ wrt (<Declaration> @wrt)
        (<Predicate> @definition) |
        (<EventSignature> @signature){local (<<ObjName>> @local) *}
        <DEFINITION> {<AXIOMS> }]
<Equality> =: (<Reference> @left) = (<Reference> @right)
<Equivalence> =: iff (<Predicate> @predicate1)
        then (<Predicate> @predicate2)
<EventClass> = <CompositeEven> | <SimpleEvent>
<EventSignature> =: (<<EventName>> @name)
        (<EventSignatureArg> @arguments) * {(: @result)
        (<Declaration> @arguments) * }
<EventSignatureArg> = <Type> | <NamedDeclaration>
<FuncClass> =: FuncClass (<FunctionSignature> @signature)
        [ <DEFINITION> | primitive ]
<FunctionalReference> =: (<<FuncName>> @function)
        (<Reference> @arguments) *
<FunctionSignature> =: (<<FuncName>> @name)
        (<Declaration> @arguments) *
        = (<Declaration> @funval)
<History> =: History : (<Predicate> @definition)
<HistoryElement> =: <<historyElement>> (<<ObjName>> @object)
        (<<NUMBERP>> @index)
        (<Predication> @event)
<ImplicitArg> =: ImplicitArguments agent : (<<ObjName>> @agent)
        last : (<<ObjName>> @last)
<Inclusion> =: Inclusion (<<SystemName>> @system)
```

```
          <INCLUDING> <RENAMING>
<Inequality> =: Neq (<Reference> ©left)
          (<Reference> ©right)
<Integer> =: (<<FIXP>> ©item)
<LetConstruct> =: let (<LetNaming> ©namings) *
<LetNaming> =: (<<LetName>> ©letname) (= ©operator)
          (<Reference> ©reference)
<LocalNaming> =: (<<ATOM0>> @name) =
          (<FunctionalReference> ©reference)
<NamedDeclaration> =: (<<ObjName>> ©name) :
          [ (<<a>> ©article) {(certain ©arbitrary) }
          (<<ObjName>> ©class) {suchThat (<Predicate> ©modifier)* }|
          (<<NamedDeclarationKEY>> ©composition)
          [ (<<ObjName>> ©type) |
          (<AlternativeType> ©type)
          ] {suchThat (<Predicate> ©modifier)*}
          | (<<CompositionOf>> ©composition)
          (<Declaration> ©type) * | (<<ObjName>> ©class) ]
<NegatedPredicate> =: Not (<Predicate> ©predicate)
<Number> =: (<<NUMBERP>> ©item)
<ObjClass> =: (<<ObjClassKEY>> ©subtype)
          (<NamedDeclaration> ©signature)
          <GENERALIZATIONS> {(<Default> ©default)}
          {(<NamedDeclaration> ©form/content/frame)}
          {(<NamedDeclaration> ©content)}
          <DEFINITION> <CONST> <VAR>
          <STATUS> <PROPERTY> <AXIOMS>
<ObjFunValSet> =: ValSetOf (<ObjFunctionalReference> ©objfunref)
          suchThat (<Prcdicatc> ©characteristic)
<ObjFuncDecl> =: {(subClassconst ©subclassconst) }
          (<<FuncName>> ©function)
          : [ <<a>> {( certain ©arbitrary) }(<<ObjNaine>> ©range) |
          (<<ObjFuncDoclKEY>> ©composition)
          [ (<Type> ©range) | (<AlternativcType> Grange)]
          | (<<ObjName>> ©range)
          {; definition (<Predicate> ©definition) * | primitive }
<ObjFunctionalReference> =: (<<FuncName>> ©function)
          [ (<AtomReference> ©arguments) |
          (<ObjFunctionalReference> ©arguments) ] |
          [ (<AtomReference> ©arguments) |
          (<ObjFunctionalReference> ©arguments) ] .
          (<<FuncName>> ©function)
<ObjInstance> =: Objinstance (<NamedDeclaration> ©signature)
          <CONST> <VAR> <STATUS> <PROPERTY> <AXIOMS>
<ObjPropDecl> =: (<<RelName>> ©relation)* ;
          [ definition (<Predicate> ©definition) * | primitive ]
<ObjSignature> =: (<<ObjName>> ©name)
<ObjStatDecl> =: (<<RelName>> ©relation)*;
          [ definition (<Predicate> ©definition)* | primitive ]
<Overview> =: Overview (<<ObjName>> ©name);
          <COMPONENT> <CAPACITY> <GOALS>
<Predicate> = <TemporallyConstrainedPredicate> |
          <TemporalPredicate> | <Predication> | <TemporalOrder> |
          <TemporalAssertion> | <Equality> | <ChangeFunVal> |
          <HistoryElement> | <ScopedPredicate> |
          <Repetition> | <WhileLoop> | <QuantifiedTemporalPred> | .
          <QuantifiedPredicate> | <SeeConstruct> |
          <Equivalence> | <ConditionalPredicate> | <BinaryBoolPredicate>|
          <NegatedPredicate> | <BreakPoint>
<Predication> =: (<<PredName>> ©predname)
          (<Reference> ©arguments) *
<Quantification> =: (<<QuantificationKEY>> ©quantifier)
          ((<<ObjName>> ©varORpred)) +
          {: [ setOf
          (ObjName ©type) |
          (ObjName ©type)]}
          {in
          (<Reference> ©domain) }
<QuantifiedPredicate> =: ((<Quantification> ©quantifications)) +
          (<Predicate> ©predicate)
```

```
<QuantifiedReference> =: ((<Quantification> @quantifications)) +
        (<Reference> @reference)
<QuantifiedTemporalPred> =: (<Quantification> @quantifications)
        (<<IsTemporalConnective>> @connective)
        (<Predicate> @predicate)
<Reference> = <FunctionalReference> | <Declaration> |
        <QuantifiedReference> |
        <Enumeration> | <SeqElementRef> |
        <AtomReference> | <ConstRef>
<RelClass> =: RelClass (<RelationSignature> @signature)
        {<RELTYPE> }{<RELGENERALIZATIONS> }
        {( static @kindtag) | (temporal @kindtag)}
        [ <DEFINITION> | (primitive @primitive) ] <AXIOMS>
<RelationSignature> =: (<<RelName>> @name)
        (<Declaration> @arguments) *
<Relation> =: (<<RelName>> @relname) (<Reference> @arguments) *
<Repetition> =: repeat {for (<<VarName>> @var) =
        (<Reference> @lower) to (<Reference> @upper) }
        (<Predicate> @predicate)
<ScopedPredicate> =: (<LetConstruct> @localnaming)
        (<Predicate> @predicate)
<SeeConstruct> =: see (<AtomReference> @see)
<SeqElementRef> =: Element (<Reference> @index)
        {of }(<Reference> @sequence)
<SimpleEvent> =: EventClass (<EventSignature> @signature)
        primitive {(<ImplicitArg> @implicitarg)}
        <NECESSARYCOND> <CONSEQUENCE>
        {<AXIOMS> }{<THEOREMS> }
<SimpleType> =: (<<a>> @article) {( certain @arbitrary) }
        (<<ObjName>> @class) {suchThat (<Predicate> @modifier)*}|
        (<<ObjName>> @class)
<String> =: (<<STRINGP>> @item)
<TemporalAssertion> =: (<<TemporalAssertionKEY>> @operator)
        (<Predicate> @events)
<TemporalConnective> =: ; | ! | : | & | / | or| and
```

```
<TemporalOrder> =: (<Predicate> @predicate1)([< |>] @operator)
        (<Predicate> @predicate2)
<TemporalPredicate> =: (<Predicate> @predicates)
        ((<<IsTemporalConnective>> @connective)
        (<Predicate> @predicates)) +
<TemporallyConstrainedPredicate> =: (<Predicate> @predicate)
        (<<TemporallyConstrainedPredicateKEY>> @operator)
        (<Predicate> @predicate1)
        {(<Predicate> @predicate2)) }
<Type> = <SimpleType> | <CompositionType> | <CollectionType>
<TypeDeclaration> =: (<Type> @type)
<WhileLoop> =: while (<Predicate> @whilecond)
        repeat (<Predicate> @predicate)
```

Production Rules for Keywords

```
<<a>> = a | an
<<CollectionTypeKEY>> = PairOf | BagOf | TupleOf|
        StringOf | TableOf | SequenceOf | SeqOf | Setof
<<CompositionTypeKEY>> = ArrangementOf | compositionOf | recordof
<<EventSignatureArgKEY>> = Var | Nil
<<NamedDeclarationKEY>> = PairOf | BagOf | TupleOf|
        StringOf | TableOf | SequenceOf | SeqOf | setof
<<NegatedPredicateKEY>> = NOT
<<ObjClassKEY>> = AbstObjClass | ConcObjClass |
        ObjClass | AgentClass | InfoObjClass
<<ObjFuncDeclKEY>> = CompositionOf | recordOf | pairOf | bagOf |
        tupleOf | stringOf | tableOf | sequenceOf | seqOf | SetOf
<<QuantificationKEY>> = ForSome | forall
<<RelClassKEY>> = Agentive | SocialRelation | Interaction
<<TemporalAssertionKEY>> = sometimes | always| previously
<<TemporallyConstrainedPredicateKEY>> =
        Before | When | While | Between
```

Production Rules for Pseudo Classes

```
<AXIOMS> = Axioms (<Predicate> ©axioms)*
<CAPACITY> = Capacities (<CapacityDeclaration> ©capacities)*
<COMPONENT> = Components : (ObjName ©components)*
<CONSEQUENCE> = Consequence (<Predicate> ©consequence)*
<CONST> = Const (<ObjFuncDecl> ©const)*          (<RedundObjFun> ©coi
<DEFINITION> = Definition (<Predicate> ©definition) *
GENERALIZATIONS > =
          Generalizations (<<ObjName>> ©generalizations)*
<GOALS> = Goals (<Predicate> ©goals)*.
<INCLUDING> = including (<<ObjName>> ©including)*
<NECESSARYCOND> = NecessaryCond (<Predicate> ©necessary)*
<PROPERTY> = Property (<ObjPropDecl> ©properties)*
<RELGENERALIZATIONS > =
          Generalizations (<<RelName>> ©generalizations)*
<RELTYPE> = type (KEY ©reltype)
<<RENAMING>> = renaming (<NamePair> ©namepair)*
<STATUS> = Status (<ObjStatDecl> ©status)*
<THEOREMS> = (<EventTheorems> ©theorems))
<VAR> = Var (<ObjFuncDecl> ©var)*
          (<RedundObjFun> ©var)*
```

Bibliography

[Allen] Allen, J., Towards a general theory of action and time. *Journal of Artificial Intelligence*, vol. 23, No. 2, July, 1984.

[Balzer and Goldman] Balzer, R.M., and N.M. Goldman, Principles of good software specification and their implications for specification languages. *Proceedings of the Specification of Reliable Software Conference*, Boston, Massachusetts, April, 1979, pp. 58-67.

[Barwise and Perry] Barwise, J. and John Perry, *Situations and Attitudes*, MIT Press, 1983.

[Barstow] Barstow, David, *Knowledge-based Program Construction*, New York, North Holland, 1979.

[Birtwistle, Dahl, Myhrhaug, and Nygaard] Birtswistle, O.J. Dahl, Myhrhaug, and K. Nygaard, *Simula Begin*, 1973.

[Blau 1981a] Blau, U. Abstract objects. *Journal of Theoretical Linguistics* vol. 8, 1981. pp. 101-130.

[Blau 1981b] Blau, U. Collective objects. *Journal of Theoretical Linguistics* vol. 8, 1981. pp. 131-144.

[Bobrow and Collins] Bobrow, Daniel G. and Allen M. Collins, (Eds.) *Representation and Understanding: Studies in Cognitive Science*, New York, Academic Press, 1975.

[Bobrow and Winograd] Bobrow, D.G., and Winograd, T., An overview of KRL, a Knowledge Representation Language, *Cognitive Science* vol. 1, no. 1, January, 1977, pp. 3-46.

[Bobrow and Stefik] Bobrow, D.G., and M. Stefik, *The LOOPS Manual.* Memo KB-VLSI-81-13, Xerox Palo Alto Research Center, revised version, 1983.

[Bobrow 1984] Bobrow, D.G.(ed.), *Journal of Artificial Intelligence,* Special volume on qualitative reasoning, Vol. 24, **No. 1-3, 1984.**

[Brachman] Brachman, R., What's in a concept: structural foundations for semantic networks. BBN report 3433, October **1976.**

[Brand] Brand, Myles, Introduction: Defining "causes", in M. Brand (ed.), *The Nature of Causation,* University of Illinois Press, 1976.

[Bunge] Bunge, Mario, *Causality and Modern Science,* Dover, New York, 1979.

[Burks] Burks, Arthur, The logic of causal propositions. *Mind,* 60, 1951.

[Burstal and Goguen] Burstal, R.M. and Goguen, J.A., An informal introduction to specifications using Clear, in R.S. Boyer and T. Strother Moore, (ed.), *The Correctness Problem in Computer Science,* Academic, 1981, pp. 185-212.

[Clarke] Clarke, L., Symbolic evaluation methods in program analysis, in Muchnick S.S., Jones, N. D.(ed.), *Program Flow Analysis.* Prentice-Hall Inc., 1980.

[Cohen] Cohen, Don, A forward inference engine to aid in understanding specifications, *Proceedings of the National Conference on Artificial Intelligence,* 1984.

[Collingwood] Collingwood, R.G., On the so-called idea of causation. *Proceedings of the Aristoltelian Society,* 38, 1938.

[Davidson 1967] Davidson, Donald, Causal relations, *Journal of Philosophy,* 64, 1967.

[Davidson 1980] Davidson, D., *Essays on Actions and Events.* Clarendon Press. Oxford, 1980.

[Davis et al.] Davis, R., H. Shrobe, et al., Diagnosis based on description of structure and function. *Proceedings of the National Conference of Artificial Intelligence,* Pittsburgh, 1982. pp.137-143

[De Klcer and Brown] Dc Kleer, Johan and John Sccly Brown, A qualitative physics based on confluences, *Journal of Artificial Intelligence,* Special volume on qualitative reasoning, Vol. 24, No. 1-3, 1984.

[Dc Kleer 1984b] De Kleer, Johan, How circuits work, *Journal of Artificial Intelligence,* Special volume on qualitative reasoning, Vol. 24, No. 1-3, 1984.

[Dijkstra] Dijkstra, E., *A Discipline of Programming*, Prentice Hall, 1976.

[Ducasse] Ducasse, C.J., On the nature and the observability of the causal relation, *Journal of Philosophy*, 23, 1926.

[Erman] Erman, Hearsay-III, *Proceedings of International Joint Conference of Artificial Intelligence*, 1981, pp. 409-413

[Fahlman] Fahlman, Scott. E., *NETL: A System for Representing and Using Real-World Knowledge*, Cambridge, MIT Press, 1979.

[Fikes and Nilsson] Fikes, R. and Nilsson, N, STRIPS: A new approach to the application of theorem proving to problem solving. *Journal of Artificial Intelligence*, Vol.2, No.3, pp. 189-208.

[Fillmore] Fillmore, C., The case for case, in Bach and Harms (Ed.), *Universals in Linguistic Theory*, Chicago, Holt, 1968, pp. 1-90.

[Forbus] Forbus, K.D., Qualitative process theory, *Journal of Artificial Intelligence*, vol. 24, 1984.

[Genesereth et al.] Genesereth, M., Russell Greiner, and David E. Smith, *MRS Manual*, Stanford Heuristic Programming Project Memo HPP-80-24, 1981.

[Genesereth 1982] Genesereth,M.R., Diagnosis using hierachical design Models. *Proceedings of the National Conference on Artificial Intelligence*, Pittsburgh, 1982, pp. 278-283.

[Genesereth 1984] Genesereth, M., The use of design descriptions in automated diagnosis, *Journal of Artificial Intelligence*, Special volume on qualitative reasoning, Vol. 24, No. 1-3, 1984.

[Goldberg and Robson] Goldberg, Adele and David Robson, *Smalltalk-80: The Language and its Implementation*, Reading, Addison Wesley, 1983.

[Greenspan] Greenspan, Sol J., *Requirements modeling: A knowledge representation approach to software requirements definition*, Technical report CSRG-155, University of Toronto, 1984.

[Guttag, Horning and Wing] Guttag, J., Horning, J., and J. Wing, Some notes on putting formal specifications to productive use. Xerox PARC Technical report CSL-82-3. June 1982.

[Hayes 1974] Hayes, Patrick J., Some problems and non-problems in representation theory. *Proceedings of the A.S.I.B Summer Conference*, Essex University, 1974, 63-79.

[Hayes 1977] Hayes, Patrick J., In defence of logic, *Proceedings of the Fifth International Joint Conference on Artiicial Intelligence*, (1977), 559-565.

[Holbaek-Hanssen Handlykken and Nygaard] Holbaek-Hanssen E., Handlykken P. and Nygaard K., System Description and the Delta Language, Delta project Report No.4, Norwegian Computing Center, February, 1977.

[Holt 1971] Holt, A., Introduction to occurence systems, in Jacks (ed,), *Associative Information Techniques*, Elsevier, 1971.

[Holt 1985] Holt, Anatol, Coordination technology and Petri nets, unpublished paper, 1985.

[Hopcroft and Ullman] Hopcroft, John E., and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Reading, Addison Wesley, 1979.

[Horning and Guttag] Horning, J. and Guttag, J., Formal specification as a design tool. *Proceedings of Principles of Programming Languages Conference*, Las Vegas (1980) pp. 251-261.

[Katz and Fodor] Katz, J. and Jerry Fodor, The Structure of a Semantic Theory, *The Structure of Language*, (ed.) Katz and J.Fodor, Prentice-Hall.1964

[Kuipers] Kuipers, B., Commonsense reasoning about causality: Deriving behavior from structure, *Journal of Artificial Intelligence*, vol. 24,1984.

[Labov] Labov, William, The boundaries of words and their meanings, in C-J. N. Bailey and Roger Shuy (ed.), *New Ways of Analyzing Variation in English*, Georgetown Univ., 1973.

Meyer]

[Lamport] Lamport, Leslie, Specifying concurrent program modules, *ACM Transactions on Programming Languages and Systems* vol. 4, no. 2, April 1983, 190-222.

[Lansky and Owicki] Lansky, Amy and Susan S. Owicki, Gem: a tool for concurrency specification and verification, *Proceedings of the Conference on Principles of Programming Languages*, 1983.

[Lauer, Torrigiani, and Shields] Lauer, RE., P.R. Torrigiani and M.W. Shields, COSY: a system specification language based on paths and processes, *Ada Informatica,* **12** (1979), 109-158.

[Lehnert] Lehnert, W., Representing physical objects in memory, Research Report # **131,** Yale University, Department of Computer Science, May **1978.**

[Lynch] Lynch, A.N. *Concurrency control for resilient nested transactions.* Technical Report 285, Laboratory for Computer Science, **MIT.**

[Mackie] Mackie, J.L., *The Cement of the Universe,* Clarendon Press, **1980.**

[Manna] Manna, Zohar, *Mathematical Theory of Computation.* McGraw Hill. 1974.

[Maturana and Varela] Maturana, H. and F. Varela, Autopoietic systems, Report No. 9.4, Biological Computer Laboratory, Dept. of Electrical Engineering, University of Illinois, 1975.

[McCarthy and Hayes] McCarthy, John, and Patrick J. Hayes, Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie (Eds.), *Machine Intelligence* 4, Edinburgh, 1969, pp. 463-502.

[McCarthy 1977] McCarthy, John, Epistemological problems of artificial intelligence, *Proceedings of International Joint Conference on Artificial Intelligence,* 1977. pp. 1038-1044.

[McDermott] McDermott, Drew, A temporal logic for reasoning about processes and plans, Technical report 196, Dept. of Computer Science, Yale University, 1981.

[Minsky] Minsky, M., A framework for representing knowledge, In P. Winston, (Ed.), *The Psychology of Computer Vision,* McGraw-Hill, 1975.

[Mitchell] Mitchell, T. M., Version spaces: An approach to concept learning, Report No. STAN-CS-78-711, Computer Science Dept., Stanford University. 1978.

[Moore] Moore, R., Reasoning about knowledge and action, unpublished Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, MIT, 1979.

[Mourelatos] Mourclatos, A.P.D., Events, processes, and states, *Linguistcs and Philosophy,* No. 2, 1978.

[Nagel] Nagel, Ernest. Types of causal explanations in science, in D. Lerner (ed.), *Cause and Effect*, The Free Press, 1965.

[Owicki] Owicki, S.S., Axiomatic proof techniques for parallel programs. TR 75-251, Department of computer science, Cornell University, July 1975.

[Peterson] Peterson, James L., Petri Nets, *Computing Surveys*, Vol. 9, No. 3, September 1977.

[Rieger and Grinberg] Rieger, Chuck and Milton Grinberg, The declarative representation and procedural simulation of causality in physical mechanisms, *Proceedings of International Joint Conference on Artificial Intelligence*, Cambridge, MA, 1977.

[Rosch] Rosch, E., Cognitive representations of semantic categories, *Journal of Experimental Psychology: General*, 1975, 104, pp. 192-233.

[Russell] Russell, Bertrand, On the notion of cause. *Proceedings of the Aristoltelian Society*, 13, 1913.

[Sacerdoti] Sacerdoti, Earl, The non-linear nature of plans, in *Advance Papers of the fourth international conference on artificial intelligence*, Tsibilisi, 1975.

[Schank 1973] Schank, Roger, *The Fourteen Primitive Actions and their inferences* (AIM-183). Stanford, California, Stanford University, Computer Science Department, 1973.

[Schank and Abelson] Schank, Roger and Robert Abelson, *Script, Plans, Goals and Understanding*, John Wiley & Son, Hillsdale, New Jersey, 1977.

[Schwartz] Schwartz, Jacob, *On programming: an interim report on the SETL project*; Installment I: Generalities, NYU Courant Institute, February 1973

[Simon] Simon, Herbert A., Causal ordering and identifiability, Daniel Lerner (ed.), *Cause and Effect*, The Free Press, 1965.

[Smith] Smith, B.C., Reflection and semantics in a procedural language, Technical Report 272, Laboratory for Computer Science, MIT. 1982

[Sosa] Sosa, Ernest, Introduction to causation and conditionals, Ernest Sosa, (ed.) *Causation and Conditionals*, Oxford University Press, 1975.

[Sussman] Sussman, Gerrald J., *A computational model of skill acquisition,* Amsterdam, North Holland, 1976.

[Swartout] Swartout, William R., The Gist behavior explainer, ISI/RS-83-3, Los Angeles, University of Southern California, 1983.

[Taylor Richard] Taylor, Richard, Causation, *The Monist,* 47, No.2 (1963).

[Taylor 1981] Taylor, R.N., Complexity of Analyzing Concurrent Programs, Department of computer science TR #DCS-9-IR, University of Victoria, 1981.

[Taylor 1983] Taylor, R.N., A General-purpose algorithm for analyzing concurrent programs. *CACM,* May 1983, pp. 362-377.

[Wilks] Wilks, Y., Good and bad arguments about semantic primitives, D.A.I. Research Report No. 42, University of Essex, May 1977.

[Winograd 1975] Winograd, Terry, Breaking the complexity barrier (again), *ACM SIGPLAN Notices,* vol. 10, no. 1, January, 1975, 13-30.

[Winograd 1978] Winograd, Terry, On primitives, prototypes, and other semantic anomalies, *Proceedings from Theoretical Issues in Natural Language Processing IL* University of Illinois at Champaign-Urbana, 1978, pp. 25-32.

[Winograd 1979] Winograd, Terry, Beyond programming languages, *CACM,* vol. 22, no. 7, 1979, pp. 391-401.

[Winograd 1984] Winograd, T., *Aleph, a system specification language,* Stanford technical report, in preparation.

[Wolterstoff] Wolterstoff, N., *On Universal.* University of Chicago Press. 1970

[Wood] Wood, W., What's in a link,in D.G. Bobrow and A. Collins (ed.), *Representation and Understanding,* New York, Academic Press, 1975, pp. 35-82.

[Wright] Wright, Larry, *Teleogical Explanations,* University of California Press, 1976.

[Xerox] Xerox Corp., *Interlisp Reference Manual,* Xerox Corporation, 1983.

[Yolton] Yolton, John W., *Metaphysical Analysis,* University of Toronto Press, 1967.

[Zave] Zave, Pamela, Executable requirements for embedded systems, *Proceedings of 5th International Conference on Software Engineering,* 1981.

Index