Inductive Modeling of Discrete-event Systems: A TMS-based Non-monotonic Reasoning Approach

by

Hessam Sarjoughian

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

1995

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Hessam Sarjoughian

entitled Inductive Modeling of Discrete-event Systems: A TMS-based Non-Monotonic Reasoning Approach

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of ______________________

	4/25/95 Date
R. François E. Cellier	4/25/95 Date
B P Zeigler	4/25/95 Date
A Terry Bahill	4-21-95 Date
	Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

R. François E. Cellier / B P Zeigler Dissertation Director	4/25/95 Date

STATEMENT BY AUTHOR

SIGNED:______________________________

ACKNOWLEDGMENTS

Many people have contributed to this work, not all of whom I can name. I am especially indebted and grateful to my co-advisors: François Cellier and Bernard Zeigler. They have contributed greatly by providing much invaluable help and support, even while I was working in the "real-world." I have had the fortune to be in an environment where my advisors allowed me to follow my interest and instinct while keeping me from straying too far! To them, my special thanks. I also wish to express my sincere appreciation to Terry Bahill, my minor advisor, as well as Jerzy Rozenblit and Fei-yue Wang for helping my studies and also serving as committee members.

Outside the University of Arizona, Takao Asahi's involvement, during his sabbatical here, was quite helpful. Advice from Alex Meystel and Anil Nerode proved to be very valuable. To them, my sincere thanks. Also, I have benefitted greatly from the works of Matt Ginsberg, John McCarthy, Nil Nilsson, and Wayne Wymore.

Finally, partial support of this research under NSF grant No. ASC-9318169 is acknowledged.

Needless to say that the goodwill of *certain others* made my studies possible. My special thanks to them.

TABLE OF CONTENTS

TABLE OF CONTENTS — *Continued*

TABLE OF CONTENTS — *Continued*

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In this presentation, we propose a framework for inductive modeling of discrete-event systems called *Discrete-event Inductive Reasoner, DIR.* It is based on *systems theory* and *non-monotonic logic.* We present a new representation for a finite set of discrete-event observed input/output time segments called *Iterative IOFO* (Input/Output Function Observation) Specification. We also introduce a novel use of non-monotonic logic allowing DIR to make *tentative* decisions. With the inclusion of additional data, non-monotonic logic ensures that any of its prior decisions that becomes violated is retracted properly. Due to the underlying features of non-monotonic reasoning, DIR supports incremental refinement/extension of the model iterative IOFO specification.

To implement the DIR, we map the iterative IOFO into a logic-based representation suitable for *Logic-based Truth Maintenance System*, a form of non-monotonic reasoning mechanism. *Abstraction mechanisms* are defined that are capable of predicting unobserved input/output time segments, given some existing IO segments and some assumptions. The systems theory framework enables us to develop the means to ensure the appropriate use of abstractions. In this way, the model is incrementally extended by predicting and retaining unobserved IO segments in a well-defined fashion.

Also, we discuss an implemented prototype of DIR called *Logic-based Discrete-event Inductive Reasoner, LDIR.* Two examples are used to discuss LDIR's features. We give some heuristic metrics for quantitative evaluation of LDIR's predictions. We provide general guidelines for the evaluation of DIR and place the methodology within existing inductive modeling approaches. We conclude with some shortcomings of our approach and speculate on future research directions.

Chapter 1 Introduction

1.1 Motivations

We are always searching for new ways to solve problems and tailor the world to our needs. Modeling is a tool employed in all branches of science and engineering to achieve this. It is used for describing the behavior of dynamical systems ranging from electrical circuits to the interactions in human societies, as well as static functions such as the relationship between air density and altitude above sea level on planet Earth or that between the lending rate and several economic factors such as inflation and unemployment.

The theory of modeling will evolve as long as it continues to help us gain an improved understanding of the environment we live in, or find better answers to our questions and solutions to our problems. As our expectations change, the techniques of modeling that we employ are constantly challenged. Hence we should not be surprised to see that no individual modeling methodology ever fully satisfies the changing expectations of all of its users. Methodologies either prosper or become obsolete depending on their ability to meet the challenges facing them. If we want to continue exploring new problems — and studying old problems in new ways — we must find more powerful and increasingly flexible modeling methodologies.

As a case in point, a movement began in the 1970's toward developing modular and hierarchical modeling paradigms. They have since proven to be indispensable! Formalists and experimentalists alike have conducted research to better understand their roots and ramifications and consequently develop methodologies and tools that adequately support them [HEO93, Wym93, Zei84, Zei90, Fis93, Cel91].

A more recent movement deals with complex, large-scale systems by attempting to develop additional capabilities that can use highly parallel computing platforms

[MRS88, CZ94]. These new developments are a testimony to the ever-evolving demands for more powerful modeling methodologies.

Another movement in modeling has been targeted toward building (modest) intelligence into models. Over the past two decades, the modeling practioners at large have witnessed progress in building some intelligence into models expressed using either deductive, inductive, or abductive paradigms. Several research communities have been involved in building intelligent models of dynamical systems [Dav84, For84, HM85, Mey85, dKW87, GN87, Ham91, Rei87, Pea88, Zei90, FF91, Cel91, Fis93, RR93].

The dynamics (or behavior) of a system can be encoded in several languages, such as differential equations and logic formulas. In particular, a system can be classified as either dynamic (time-dependent) or static (time-independent). In this work, I am concerned with dynamic systems only. A dynamical system may be modeled using differential equation, discrete-time, or discrete-event formalisms [Cel91, HM85, Sho88, Wym93, Zei76, Zei84]. Each of these formalisms is associated with a particular time base. Systems represented (modeled) using either a differential equation or a discrete-event formalism have a continuous time base, while those employing a discrete-time formalism have a discrete time base. Furthermore, dynamical systems can be viewed as either time-invariant or time-variant [Cel91, Wym93, Zei76], deterministic or stochastic [Wym93, ZD79, Zei76], as well as causal or non-causal [Mac74, Pea88, Sho88, Cel91, Sup73, ZD79, Zei76]. The notion of time-invariance *vs.* time-variance, associated with a dynamical system, determines whether its behavior depends on a fixed time point or not. A deterministic model is not subject to uncertainty as a stochastic model would be. The causal behavior of a model asserts that its previous behavior influences its present and future behavior. The past or present behavior of a noncausal model, however, may depend on its future behavior. In this dissertation, I am concerned with a class of dynamical systems, which are time-invariant, deterministic, and causal.

Furthermore, a modeling methodology can be viewed as either deductive, abductive, or inductive depending on whether outputs, inputs, or structure of a system are the subject of inquiry, respectively [Cel91, GN87, Zei76, Zei84, Wym93]. The deductive modeling paradigm computes outputs of a system based on its structure (principles governing its behavior) and some valid input sets. The abductive modeling paradigm predicts a system's inputs given its structure and some outputs. The inductive paradigm *attempts* to predict a system's behavior from input and output data sets without any reference to the system's structure.

We can think of inductive modeling as either trying to identify a *structure* for the system or some set of *generator* input/output pairs. The former tries to summarize the behavior of the system in a closed-form. The latter attempts to identify a minimal set of input/output pairs from which a system's behavior can be obtained by means of composition.

Modeling is generally viewed as a creative activity that relies on complex and intricate reasoning processes. Presently, almost the entire task of creating a model is the responsibility of the modeler. Relying on diverse sources of knowledge, modelers create, implement, and eventually try to validate their models. The act of reasoning is such an integral part of these stages that sometimes modelers are hardly *aware* of it.

In recent years, improved measurement and data storage techniques have led to an avalanche of data becoming available that characterize a large variety of processes or systems. Consequently, there is a need for methods capable of reasoning about these data. That is, a modeler studies some data and tries to devise a model that explains them. As the amount of available data continues to increase rapidly, a modeler becomes overwhelmed with the amount of information. Hence it is imperative to seek methodologies that can take active part in understanding the data. Such a methodology would emulate the tasks of one or more modelers as we know them today. I call a modeling methodology that supports this kind of modeling *non-standard*.

Obviously, at the present time and for a very long time to come (if ever), we cannot hope for a model that in itself would have the ability to emulate a modeler's reasoning in its entirety. Hence, it is natural to identify the simplest among all forms of model creation and attempt to *model* it. To this end, it is, of course, necessary to isolate the part that is amenable to being modeled given our present knowledge.

Generally, of the three modeling paradigms, the simplest form of reasoning is used in inductive modeling. The basis for this claim is that the inductive modeling approach attempts to create a model for a system based on observed data. That is, we reason about a system's behavior and formulate hypotheses (develop models) about how it may account for the observed behavior. A modeler trying to derive an either deductive or abductive model typically needs to specify a system's internal structure. The reasoning processes involved in deriving such an internal structure representation of a system are quite intricate and complex.

In broad terms, inductive modeling consists of identifying input/output trajectories of interest, reasoning about the observed data (which would result in some postulated relationships among them), and gaining more and more confidence in the hypothesized relationships by unsuccessfully trying to refute them through additional experiments. It is my intent to devise an approach to model the reasoning part only.

As the overall goal of this dissertation, I propose a new inductive modeling methodology that supports an explicit form of reasoning about a system's observed behavior. Given a system's observed behavior, i.e., a set of input/output trajectories, I partition them into a set of input/output segments that can then be re-composed appropriately to form the original observed behavior as well as predict output trajectories for yet unobserved input trajectories. Of course, since we can only observe, store, and process a finite amount of data, it becomes necessary to *reason* about the available data in order to account for any unobserved data.

A very useful concept in modeling is *abstraction*. Whereas modeling itself can be viewed as a process of encoding knowledge available about a system under study, abstraction can be interpreted as a process of knowledge generalization.

The flexibility provided by *multiple abstraction levels* has proven itself indispensable in modeling activities. It provides the means to study a system from different viewpoints [HH90, HM85, Kli85, Wym93, Zei76, Zei84, Zei90, Wel92]. Its utility in mathematics and physics as well as our every day activities is prevalent and indisputable. To switch from one viewpoint to another can be seen as a change in abstraction level. Often, the need for a more abstract (simplified) representation is justified on the basis of insufficient knowledge being available to reason conclusively at a level of lesser abstraction. Higher abstraction levels (corresponding to less detailed knowledge) support better-managed reasoning (in terms of tractability and complexity) at the cost of a reduced precision.

The proposed inductive modeling approach requires partitioning of input/output trajectories. It also requires reasoning about the use of input/output segments in predicting output trajectories as necessary. I use two classes of abstractions. In particular, a trajectory can be partitioned into smaller parts (segments) based on abstractions on the time base as well as the range of the input/output variables. These abstraction levels support varying degrees of data granularity. The other class of abstractions provides the basic means for reasoning about the complexity of input/output segments. For instance, we may abstract the length of an input/output segment pair, or abstract a segment's initial state. The general concept for examining the correctness of an abstraction is *homomorphism.* It establishes relationships between two models of a system [Sto73, Zei76, Wym93].

The ability to change the abstraction level based on explicit assumptions (justifying the use of a particular abstraction level) underlies the development of a non-standard modeling methodology. Obviously, a reasoning mechanism that is able to operate at varying abstraction levels, necessitated due to a lack of knowledge, has a non-monotonic character.

Modeling has many attributes [Min65, Zei76, Zei84, Kli85, GN87, Cel91, Wym93], two of which are fundamental in bringing about sound modeling methodologies. First, it can benefit greatly from being non-standard. Second, it should be built on *well-defined concepts* supported by mathematical underpinnings. Both of these attributes are essential for the development of powerful modeling methodologies.

We can categorize modeling methodologies into standard and non-standard. A model generated using a *standard methodology* cannot in itself emulate the tasks of a modeler. Whereas these models may contain varying forms of decision making, they are, in general, not subject to revision. For example, models devised based on formulations of deductive paradigms such as differential-equation specified systems, discrete-time specified systems, and discrete-event specified systems [Cel91, Fis93, Kli85, Zei76, Wym67] provide the means for modelers to create models. They are not, in themselves, models of a modeler.

An ideal *non-standard modeling methodology*, on the other hand, would support creating models that would emulate a modeler's activities. While this approach has to possess all the capabilities afforded by the standard methodologies, it should, in addition, possess reasoning capabilities similar to our own. This approach, unlike those devised according to standard methodologies, may not always behave correctly, just as a modeler hardly ever creates a model, the first time around, that does not have to be revised. Therefore, non-standard methodologies must provide means for the integration of reasoning mechanisms into models, such that their incorrect decisions are correctable automatically by themselves.

Within deductive modeling paradigms, progress has been made to support the creation of models that have some self-assessment capabilities, i.e., that know something about themselves [Zei84, GN87, BZR89, Cel91, Zei90, Fis92]. However, these approaches provide no explicit reasoning mechanisms to handle their own incorrect decisions. It is important to observe that, whereas non-standard methodologies are prone to committing (hopefully correctable) *mistakes*, standard methodologies are expected to always make the *correct* decision right away.

To develop a non-standard inductive modeling methodology, I draw from two basic disciplines: Mathematical Systems Theory [ZD79, MT75, Kli69, Wym67, RKA69] and Artificial Intelligence [GN87, Ric83, Gin93, Nil80]. The former provides a framework for representing and studying dynamical systems. The latter supports explicit types of reasoning using declarative knowledge representation.

Although Systems Theory forms the foundation for all system representations, it is, by itself, insufficient to support modeling of complex and interconnected systems [Kli85, Wym93, Zei84]. As a result, several modeling methodologies have been developed [Zei76, Elm78, Zei84, Zei90, Kli85, Wym93] that build on Systems Theory, but add several essential components to it. In this work, I shall use a hierarchy of system specifications with abstraction levels, starting from purely observed input/output behavior and ending with the coupling of highly-structured systems [Zei76, Zei84].

As stated earlier, non-standard methodologies need to deal with changing abstraction levels expressible in declarative form. Traditionally, reasoning has been within the confines of *declarative* representations due to its rich semantics and powerful means for manipulation of descriptive knowledge [GN87, McC90, Dav90, Nil91]. In representing abstraction levels for *explicit* reasoning, I take the *logical approach* to knowledge representation [GN87], as opposed to other approaches, such as semantic nets or rule-based systems [Ric83].

To carry out the type of reasoning that is necessary for our purposes, I make use of the so-called *non-monotonic reasoning* approach. Since the late 1970's, several such approaches (formalized and unformalized) have been developed [Pol75, Doy79, McC80, McA80, MD80, Pea88, Rei80, Sho88, Bre91, MT93]. In my work, I use a variant of Truth Maintenance Systems [Doy79] called Logic-based Truth Maintenance System [McA80, FdK93]. It supports the type of non-monotonic reasoning that we are interested in.

Now, we can view standard and non-standard modeling methodologies in terms of a block diagram with three components: *language, reasoning,* and *abstraction.* Figure 1.1 depicts how a modeling methodology can be created using these components put together appropriately. In particular, a modeling methodology defines its own governing principles to ensure a well-defined discipline. According to what I have said so far, we have a non-standard methodology when only the reasoning component and the language component are used. By including the abstraction component with the other two components, we have a non-standard modeling methodology.

I wrote earlier about the need for a formal approach to modeling. The development of modeling methodologies is influenced by the fact that modeling is a conglomerate of various techniques put together to capture a particular viewpoint of a system. However, modeling can be (and should be) viewed as a discipline that seeks to formalize its existence based on solid mathematical foundations. To ignore the necessity of formalized modeling methodologies, puts the discipline at

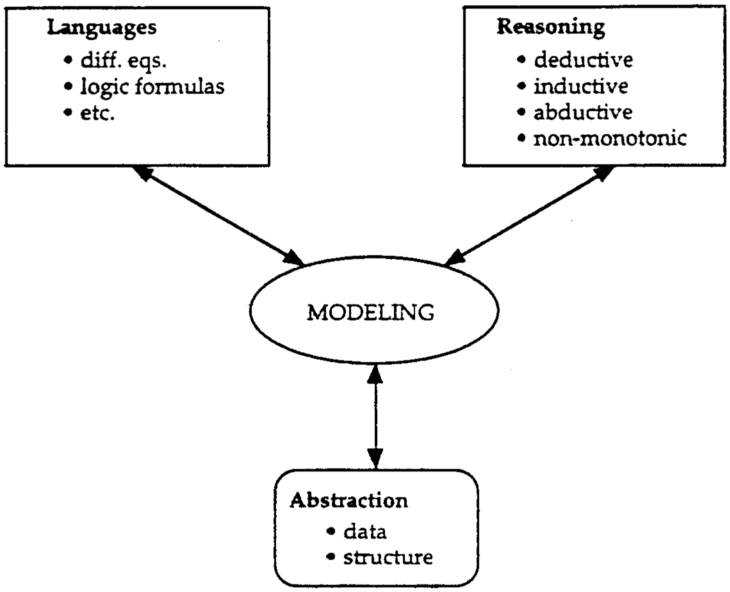


Figure 1.1, Components of Modeling methodologies

risk. Using a rigorous theoretical approach supports validation and verification of models via mathematical tools and offers several other important advantages [Zei76, Zei84, Kli85, GN87, MEZ89, Wym93].

An occasional lack of preciseness, however, may be unavoidable, and may, in fact, be beneficial at times. It is furthermore not endemic to the modeling community alone. In the field of Artificial Intelligence, spirited debates between formalists and experimentalists have been going on over the same fundamental issues. The following is an excerpt from formalists' point of view:

> *... it is nevertheless our opinion that the important new results in AI will be achieved by those researchers whose experiments are launched from the high platform of solid theory*, page *viii* [GN87].

I take the point of view that the research conducted by experimentalists is equally important as that conducted by formalists. Tangible results in science are generally brought about by both the experimentalists and the formalists. It is not always the case that research can take off from well-defined theoretical underpinnings. Sometimes theories are developed years after the experimentalists have demonstrated their own results [SS77, Doy79, McA80, MRS88, dK86a, RdK87]. Nevertheless, the development of new methodologies, as well as the growth of existing ones, may severely suffer from a lack of solid theoretical foundations, threatening the existence of a stable base for future research.

1.2 Goals

In view of what has been said thus far, I have chosen the *discrete-event* representation paradigm as the basis of my investigation, since high-level model abstraction is essential for having any hope of tractability within inductive modeling paradigms. Furthermore, it is my belief that non-standard modeling methodologies cannot entirely be built on deductive modeling paradigms, thus the need for an *inductive* approach. This is mainly due to the observation that inductive modeling is an important stepping stone toward creating deductive models. I am also interested in dynamical systems (i.e., those that have states.) Henceforth, this work is concerned with inductive, discrete-event system representations of dynamical systems. Reasoning about them is accomplished with the aid of a non-monotonic paradigm expressed in the language of logic.

Hence the goal of this dissertation is to develop an approach for an inductive discrete-event modeling methodology based on non-monotonic reasoning. The breakdown of the goal of this work results in the following:

- Formulate a representation of a deterministic, time-invariant, and causal observed input/output data set.
- Identify an assumption set-I to determine data granularity and partitioned input/output data that allows the partitioning of the observed trajectories

into discrete-event segments, if necessary. Also identify an assumption set-II that facilitates reasoning at varying abstraction levels in accordance with the richness of the available data as well as the necessity for moving from one abstraction level to another.

- Develop and formalize an iterative input/output function observation specification suitable for reasoning.
- Identify the role of non-monotonic reasoning in inductive modeling.
- Define an inductive modeling methodology incorporating some simple explicit forms of non-monotonic reasoning.
- Choose/justify an appropriate non-monotonic reasoning mechanism for representation and manipulation of observed data and the assumption set-II.
- Define an architecture for the proposed inductive discrete-event modeling methodology.
- Develop the foundations of an inductive discrete-event modeling reasoner.
- Implement and test the proposed inductive modeling reasoner via an example — a shipyard with two repair stations responsible for repairing broken vessels.
- Develop some heuristic means for evaluating the proposed approach.
- Discuss the potentials, limitations, and future work for the proposed approach as well as its relation to some of the existing work.

1.3 Organization

In Chapter 2, I begin with the presentation of fundamental concepts from Systems Theory. I use the concepts of determinism, time-invariance, and causality to formalize a representation of the input/output space. Then, I describe the notion of generator segments and extend it to the input/output space. I continue with the representation of a stratification of system specifications. This chapter concludes with the discrete-event representation of I/O System (IOS) and I/O Function Observation (IOFO) specifications.

Chapter 3 begins by describing and analyzing the iterative discrete-event I/O system specification. It discusses the importance of the iterative IOS specification. I then propose a way to dissect the IOFO specification in order to define its iterative specification. I continue with a reformulation of the IOFO specification, from which its iterative I/O function observation specification is formalized. Then, based on it, a free iterative IOFO specification is derived. Next, a discrete-event representation of the IOFO specification is introduced. I also associate assumption set-I and assumption set-II with IOFO specification and iterative IOFO specification, respectively. Assumption set-I determines the granularity of data for an iterative IOFO specification. Assumption set-II allows reasoning (abstraction) of IO segments of an iterative IOFO specification in order to construct its free specification. Finally, a free-constructed IOS specification is described and compared with the free I/O function observation specification.

The discussion of AI reasoning paradigms is presented in Chapter 4. This chapter serves three purposes: (1) to provide a brief overview of some types of non-monotonic reasoning approaches, (2) to select the one that serves best our present demands and that may serve some future purposes as well, and (3) to realize the potential of inductive modeling as a new application of non-monotonic reasoning. In particular, I focus on non-monotonic reasoning within the AI territory, which uses logic as the primary means to represent knowledge and carries out the task of reasoning from a model theoretic point of view. Hence I present a brief overview of logical approaches to knowledge representation. Two primary knowledge representation languages, propositional and first-order logic, are briefly described. Then, a primitive form of knowledge representation as well as a specific inference rule (resolution principle) that has the two fundamental properties of soundness and completeness are discussed. I continue with a discussion of reasoning, specifically non-monotonic reasoning (NMR), and why it plays a significant role in inductive modeling. Some NMR approaches (e.g., Closed-World Assumption, Negation as Failure, and Circumscription) that are based on model-theoretic logic are presented. In particular, I also describe the Truth Maintenance Systems (TMS) from a somewhat formal point of view, as it has been presented in the literature. Two variations of the TMS, implemented in the Common Lisp language, are described followed by a discussion of trade-offs between them. At the end of this chapter, I list some of the present applications of NMR and discuss my application. The last section also discusses the challenges of incorporating NMR into other problem-solving methods.

Chapter 5 establishes the details of an inductive reasoner called Discrete-event Inductive Reasoner (DIR). I begin by describing the existing abstraction mechanisms which are primarily developed for systems with their internal structures known. This discussion points to the lack of a theory of abstraction for systems with their internal structures unknown. Then, I give a modified representation of discrete-event input segments as well as input/output segments. This leads to a reformulation of the iterative IOFO specification in terms of these IO segments. To provide predictability, I devise some equivalence relations as a way to abstract IO segments to derive new ones in a well-defined manner. From these equivalence relations, a set of assumption types from assumption set-II are identified. Then, I discuss how iterative IOFO can be augmented with a Logic-based Truth Maintenance System to devise DIR. First, I choose logical representation of the IO segments, then show how assumptions provide the basis for justifying the use of equivalence relations. Thereafter, I show how unobserved IO segments can be predicted in a systematic manner. The remaining elements of the assumption set-II are determined from the types of predictions allowed. I then provide a set of consistency axioms for ensuring consistency of any predicted IO segment with previously known IO segments. I continue with devising a scheme for partitioning of input trajectories. Finally, I conclude this chapter by showing how DIR (by utilizing equivalence relations, prediction schemes for unobserved IO segments, and consistency axioms collectively) can predict output trajectories for some well-defined input trajectories.

An implementation of the inductive reasoner, called Logic-based Discrete-event Inductive Reasoner (LDIR), is presented in Chapter 6. I start by describing a shipyard. The example is concerned with arrival ships in need of repair. Each instance of this example considers the arrival of some ships and their departure after they are repaired. The shipyard features two types of disciplines (repair priorities) — one is first-in-first-out and the other is priority ranking. This example serves as the primary vehicle for studying DIR. It also shows what types of systems I am dealing with. Next, I give in pseudo code the skeleton of the procedures carrying out the tasks of the LDIR. I consider 5 scenarios. All scenarios use the same input trajectory for which output trajectories ought to be predicted. For the first three scenarios, one particular set of observed IO segments are supposed. Each of these scenarios, however, considers a particular variation of the assumption set-II. The last two scenarios use another set of observed IO trajectories. Again, each of these two scenarios uses different variations of the assumption set-II. Detailed description of the LDIR implementation is illustrated for the first two scenarios. The chapter continues with a comparison of the two disciplines and point out LDIR's features. Next, two quantitative measures are provided to determine the goodness of the LDIR's predicted behavior. I also give a qualitative evaluation of LDIR. This chapter concludes with an overall evaluation of DIR compared with some other approaches.

In the last chapter, I summarize the contribution of this work, discuss some related work, and suggest directions for future work.

Chapter 2 A System Theoretic Approach to Modeling

2.1 Preliminaries

Systems theory is based on basic formal concepts such as time, a system's behavioral properties (e.g., determinism and time-invariance), and the representation of input and output trajectories and how they may be manipulated. This chapter presents these concepts and their relationships. Indeed, the definitions of such concepts are fundamental to a precise treatment of various classes of system representations (to be outlined later in this chapter) and their interrelationships.

In this chapter, I present the main definitions and concepts without regard to any specific view of time. Later, I will specialize these definitions and concepts to continuous, discrete-time, and discrete-event forms as we need them.

2.1.1 Time Base

The notion of time seems rather trivial. Yet, its mathematical treatment and even its physical understanding have been a stumbling block for centuries [Pri80]. The notion of time is an elusive one. In general, time can be either *linear* or *branching* [Ben83, EH83]. For linear time, there is only one possible future at any moment. Branching time, however, indicates that time has a tree-like (or branching) nature — a given instant of time has more than one future (i.e., there are multiple/alternative futures).

I shall assume the nature of time to be linear. Furthermore, time is assumed to [Zei76]:

1. be an Abelian group (i.e., an algebraic structure $\langle T, +, < \rangle$),

2. satisfy a linear ordering relation $<$,

3. satisfy order preservation under addition $+$, and

4. have the property of unbounded extension.

The restriction of the time base to an Abelian group is necessary for considering time-invariant systems, i.e., for being able to move back and forth time-based histories w.r.t. time. The linear ordering relation establishes the notions of past, present, and future, while order preservation guarantees the chronology of instants of time. The remaining unbounded extension property allows us to consider trajectories of finite but arbitrary length.

Since the elements of T, a set of time points, can be either real, $\Re$, or integer, $\mathcal{I}$, the time base can be either continuous or discrete. Both continuous and discrete time bases will be used in my discussions. I exclude the discussion of hybrid time bases.

I make another observation: the notions of *time points* as well as *time periods* [All84] are used in my discussion, since I shall be dealing with both discrete and continuous time bases, although usually only one of these concepts will be considered in any particular situation.

The notation used to describe a *time period* is $< t_i, t_f >$ where $\forall t \; t_i \leq t \leq t_f \Rightarrow t \in T$. The angular brackets $< .,. >$ can stand for any of four possible choices: $[\,.,.\,), (\,.,.\,), [\,.,.\,]$, and $(\,.,.\,]$, i.e., both open and closed intervals will be considered.

2.1.2 Histories

Systems are often studied by looking at their behavioral patterns which are nothing more than sets of collected data with a *chronological* pattern. A system's behavior can be phenomenologically described through a collection of time-stamped data relating to some variables of interest, often classified as its inputs and outputs. What I mean by a data set is a *time* history of a variable. Generally the variable represents some property of a system. Thus, histories have *time* as their independent variable. Notice that not all functional relationships are called "histories" — e.g., fluid viscosity as a function of temperature is not considered a "history" unless time is explicitly present. In general, a history, denoted as ω, is represented as:

$$\omega : < t_i, t_f > \longrightarrow Z$$

where Z is a set, and t_i and t_f are the initial and final instants of time associated with the set.

In the following discussions, I shall distinguish between histories based on their interpretations. If a history is expected to be partitioned into histories of smaller durations, it will henceforth be called a *trajectory.* On the other hand, if a history is assumed not to be partitioned any further and only to be used as is, it will be called a *segment.* Note that this definition is entirely based on our interpretation of histories and the limitation put on how small their *time length* (or duration) might be. Therefore, under interpretation, a history of a variable is either a trajectory or a segment, but not both [1].

The duration of a trajectory has at least one or more intermediate time instants that can be used to partition it into two trajectories, two segments, or a trajectory and a segment. In formal notation, for any trajectory ω, there exists at least one time instant t' such that $< t_i, t_f >$ can be partitioned into $< t_i, t' >$ and $< t', t_f >$, where each part of the partitioned trajectory can itself be either a trajectory or a segment. Likewise, for a segment, denoted as $\omega_{\nless,\ngtr}$, and its time

[1] In the later part of this chapter as well as in the following chapter, I shall develop some concepts based on whether a history is a segment or not. In particular, the so-called iterative specification of an algebraic structure is based on segments.

length $dom(\omega_{\not<,\not>}) =< t_i, t_f >$, there does not exist any t' such that $< t_i, t' >$ and $< t', t_f >$ can be found. The duration of a trajectory or a segment is represented as $\ell(.) \overset{\mathrm{def}}{=} t_f - t_i$. Evidently, $\ell(\omega_{\not<,\not>}) < \ell(\omega)$.

I can distinguish between several types of histories: constants, periodic, continuous (and its variations — constant piecewise, linear piecewise, etc.), and discrete event. As indicated above, the time base can be either real-valued or integer-valued. From here on, the time base for a discrete event history will be assumed real-valued.

2.1.3 Types of Histories

Having described histories of the form $\omega : < t_i, t_f > \rightarrow Z$, I can specify three types of histories based on the time base T and its corresponding set of values Z. The set Z is either the *input value* set X or the *output value* set Y. That is $\omega : < t_i, t_f > \rightarrow X$ represents *input histories*, whereas $\psi : < t_i, t_f > \rightarrow Y$ represents *output histories.* In short, I am denoting ω and ψ as input and output histories, respectively.

I can interpret a history in any of the following forms:

- *Continuous time histories:* A continuous time history is continuous at all points along $< t_i, t_f >$; i.e., it is differentiable at least once everywhere.

  $\omega : < t_i, t_f > \longrightarrow Z \qquad \text{where} \qquad t \in \Re, \forall t \;\; t_i \leq t \leq t_f$ and
  $z \in Z \Rightarrow z \in \{numerals\} \qquad \text{where} \qquad \phi \notin symbols.$

- *Piecewise continuous histories:* A piecewise continuous history is continuous at all points along $< t_i, t_f >$ except at a finite number of points. Furthermore, $\omega(t)$ is continuous between any two consecutive time points $t_1 \leq t \leq t_2$.

  $\omega : < t_i, t_f > \longrightarrow Z \qquad \text{where} \qquad t \in I, \forall t \;\; t_i \leq t \leq t_f$ and
  $z \in Z \Rightarrow z \in \{numerals, symbols\} \qquad \text{where} \qquad \phi \notin symbols.$

- *Discrete event histories:* A discrete event history has a finite number of points $\{t_i\}$ for $i = 1, \ldots, n$ at which $\omega(t_i) \neq \phi$ where ϕ denotes the "non-event symbol". Whenever an event occurs, it is identified with a symbol other than ϕ or a numeral. In other words, a discrete-event history should have a finite number of symbols other than ϕ and/or numerals associated with it.

$\omega : < t_i, t_f > \longrightarrow Z$ where $t \in \Re, \forall t \;\; t_i \leq t \leq t_f$ and $z \in Z \Rightarrow z \in \{numerals, symbols\}$ where $\phi \in symbols.$

A special case of a discrete-event history is the *empty* history, where $\omega(t) = \phi$ for all $t_i \leq t \leq t_f$. We can also have "piecewise constant histories". This is a subclass of piecewise continuous histories where for $\omega :< t_0, t_n > \longrightarrow Z$, there exists a finite number of $t_1, \cdots, t_{n-1} \in < t_0, t_n >$ and their corresponding $c_1, \cdots, c_n$ such that $\omega = \omega_{<t_0,t_1>} \circ \cdots \circ \omega_{<t_{n-1},t_n>}$ and $\omega_{<t_i,t_{i+1}>}(t) = c_{i+1}$ for all $t \in < t_i, t_{i+1} >$ $, i \in \{0, \cdots, n-1\}$.

Later I shall use the notation (Z, T) to denote the set of all histories that belong to any one of the classes defined above for given sets Z and T. That is, $\omega : < t_i, t_f > \longrightarrow Z \in (Z, T)$.

2.1.4 Composition of Histories

In the preceding section, it was said that a trajectory with duration $< t_i, t_f >$ can be decomposed into $< t_i, t' >$ and $< t', t_f >$. Depending on which of the four possible types of intervals is selected, the two neighboring histories are either disjointed, connected, or overlap at t'. In either case, I say that the two time histories are *contiguous*, not making the interpretation unique. With this in mind, any two histories (that may belong to two distinct histories), are contiguous if they satisfy the following composition operation [Zei76]. The composition operation simply concatenates two histories:

$$\omega_i : < t_0, t_1 > \longrightarrow X$$

$$\omega_j : < t_1, t_2 > \longrightarrow X$$

$$\omega(t) = \omega_i(t) \;\circ\; \omega_j(t) = \begin{cases} \omega_i(t) & \text{if } t \in < t_0, t_1 > \\ \omega_j(t) & \text{if } t \in < t_1, t_2 > \end{cases}$$

A duration $< t_0, t_1 >$ can have any of these forms: $\{(\;),\; [\;),\; (\;],\; [\;]\}$. If ω_i and ω_j overlap at t_1, i.e., if the upper interval boundary of ω_i and the lower interval boundary of ω_j are both closed, the contiguous trajectory could potentially be multivalued at t_1, which doesn't make sense for a time function. To avoid this problem, I therefore demand, for an overlapping contiguous trajectory, that $\omega_i(t_1) = \omega_j(t_1)$.

I shall furthermore define that a collection of histories is denoted as Ω such that ω or $\omega_{\not<,\not>} \in \Omega$. Given a set of histories Ω, the composition operation is said to be *closed* if for every contiguous pair ω and $\omega' \in \Omega$, $\omega \circ \omega' \in \Omega$ is true. Furthermore, the composition is said to be *associative* if, for $\omega \circ \omega' \in \Omega$ and $\omega' \circ \omega_2 \in \Omega$ being true, $(\omega \circ \omega') \circ \omega_2 \equiv \omega \circ (\omega' \circ \omega_2) \in \Omega$ always holds.

Furthermore, the set of histories that have $< 0, t >$ as their domain is denoted by Ω_0. That is, $\omega \in \Omega_0 \Leftrightarrow \omega : < 0, t > \rightarrow Z$. Thus, Ω_0 forms a semigroup[2] under the composition operation.

Since all histories in Ω_0 begin at time zero, the composition of $\omega, \omega' \in \Omega_0$ requires translating one of the histories. Let me briefly explain the translation operation. Assuming that a history can be translated, the composition operation on histories

[2] A semigroup, as compared to a group, only needs to satisfy the closure and associativity properties [Sto73].

in Ω_0 is identical to the one presented above. Note that by definition Ω_0 contains *all* histories that begin at time zero and thus satisfies the closure property, if I demand that only one of the two composed histories is allowed to be translated. The associativity property is also satisfied, as can be easily shown by induction.

However, if histories are arbitrarily translatable, as I had to demand in order to define the composition operation on histories in Ω_0, then it is not meaningful anymore to distinguish a given fixed time point as time zero. A zero time point may still be needed for algebraic reasons, but we should not put too much emphasis on where precisely the zero time point is located. Hence I shall, from now on, talk about histories that begin at a fictitious time *zero*, and thus use Ω instead of Ω_0 in the remainder of this work. Furthermore, I shall denote the set of *input histories* $\omega_1, \ldots, \omega_n$ as Ω, and the set of *output histories* $\psi_1, \ldots, \psi_n$ as Ψ. Obviously, for correctness and consistency, the output histories must have the same form and properties that characterize the input histories for any given system.

Before proceeding to the next section, I need to introduce a notation that allows me to *partition* a history into its *left-history* and its *right-history*. To this end, I shall define the operation *cut*. It partitions a history into two histories separated by a time instant $\tau \in < t_i, t_f >$. The histories to the left and to the right of τ are denoted as $\omega_{\tau>}$ and $\omega_{<\tau}$, respectively. They can be obtained using the cut operations $cut.left(.)$ and $cut.right(.)$:

$$\omega : < 0, t > \longrightarrow Z$$

$$cut.left(\omega, \tau) = \omega_{\tau>} \qquad \Longrightarrow \qquad dom(\omega_{\tau>}) = < 0, \tau >$$

$$cut.right(\omega, \tau) = \omega_{<\tau} \qquad \Longrightarrow \qquad dom(\omega_{<\tau}) = < \tau, t >$$

where $0 \leq \tau \leq t$. Of course, $\omega = \omega_{\tau>} \circ \omega_{<\tau}$ where $< . , . > \in \{[.,.], [.,.), (.,.]\}$.

I am assuming $\omega_{<\tau}$ and $\omega_{\tau>}$ to have identical interval types. Of course, the composition property becomes undefined at a time instant τ when $< . , . > = (. , .)$. It is also possible that $\ell(.) = 0$ for one of the histories.

To translate a history to a time point located either in the future or in the past, we need an operator. I define a unary operator called *translation*, denoted as $TRANS_\tau$, which translates a history along its time base by τ either to the past ($\tau < 0$) or to the future ($\tau > 0$) with respect to the initial time point [Zei76].

Translation does not change the algebraic structure of a history, thus: $TRANS_\tau : (Z,T) \rightarrow (Z,T)$. Given a history $\omega : < t_i, t_f > \rightarrow Z$ to be translated by τ: $\omega' = TRANS_\tau(\omega)$. Then: $\omega' : < t_i + \tau, t_f + \tau > \rightarrow Z$. The operation $TRANS_\tau(\omega)$ must satisfy $\omega(t) = \omega'(t + \tau)$ for all $t \in < t_i, t_f >$.

The set Ω is said to be *closed under translation* if $TRANS_\tau(\omega) \in \Omega$ for all $\omega \in \Omega$ and for all $\tau \in T$. Closedness under translation implies the unbounded extension property that was demanded earlier in this chapter, as can be shown easily by induction.

Both the *cut* and *translation* operations are applicable in the left and the right directions. The two operations can be combined to define various forms of the operation *cut.shift* [AZ93]. In this text, the operation *cut.shift* denotes a *cut.right* operation followed by a $TRANS_\tau$ operation in the left direction to move the left end of the right-history back to the fictitious zero point. Given $\omega : < 0, t > \rightarrow Z$, we have:

$$\begin{aligned} cut.shift_\tau(\omega) &= TRANS_{-\tau}(cut.right(\omega, \tau)) \\ &= TRANS_{-\tau}(\omega'), \qquad \omega' : < \tau, t > \longrightarrow Z \\ &= \omega'', \qquad \omega'' : < 0, t - \tau > \longrightarrow Z \end{aligned}$$

Hence:

$$cut.shift_{\tau}(\omega) : < 0, \ell(\omega) - \tau > \longrightarrow Z, \qquad \text{such that}$$

$$cut.shift_{\tau}(\omega)(t) = \omega(t + \tau) \qquad \text{for} \qquad t \in < 0, \ell(\omega) - \tau > .$$

where τ is the time instance at which the operation cut takes place and the amount by which the part of the history to its right is translated to the left w.r.t. t. Having defined the *cut.right* and *cut.shift* operations, we have the following proposition:

Proposition 1 *The set of histories Ω_0 satisfies: [Zei76]:*

- *closure under composition,*
- *closure under the cut.shift operation, and*
- *closure under the cut.left operation.*

2.1.5 Time-Invariance and Causality of Input/Output Histories

Systems are often assumed to be time-invariant and deterministic, since systems that satisfy these properties can be treated with more rigor and preciseness than with those that lack such properties. The discussion of time-invariance and determinism is often given within the context of a system's structure as opposed to its observed behavior. Appropriate restrictions are put on a set of differential equations describing their underlying features. In my discussions, the behavior of a time-invariant and deterministic system (i.e., its histories) is a direct consequence of its structural representation, and hence time-invariance and determinism of histories follow from the same properties of the underlying system that generates these histories.

Nevertheless, it is also useful to define time-invariance and determinism directly as properties of input/output histories, rather than indirectly through the properties of a system's internal structure that accounts for the observed input/output histories. Let me give some definitions.

Definition 1 *A set of input/output histories:*

$$IOspace = \{(\omega, \psi) \mid (\omega, \psi) \in (X, T) \times (Y, T), \qquad dom(\omega) = dom(\psi)\}$$

is called an $IOspace$.

The domain of each pair $(\omega, \psi) \in IOspace$ can be restricted to $< 0, t >$ for some $t > 0$. Hence:

Definition 2 *A set of input/output histories beginning at time zero*

$$\begin{aligned} IOspace_0 = \ & \{(\omega, \psi) \mid (\omega, \psi) \in IOspace, \\ & dom(\omega) = dom(\psi) =< 0, t > \qquad for\ some \qquad t > 0\} \end{aligned}$$

is called an $IOspace_0$.

I can use the projection function [Wym67] over the $IOspace_0$ onto the first and second coordinates to obtain Ω_0 and Ψ_0, respectively. Hence $PJN(IOspace_0, 1) = \Omega_0$ and $PJN(IOspace_0, 2) = \Psi_0$.

To discuss the time-invariance and causality notions in the context of composite input/output histories, the operators *cut.right*, *cut.left*, and *translation* need to be well-defined. My interpretation of well-definedness of the above operators is:

Definition 3 *If the operators cut.left, cut.right, and translation (hence cut.shift) generate time-invariant input/output histories from a set of time-invariant input/output histories, then they are said to be well-defined.*

In other words, these operations must not affect the time-invariance characteristic of their operands (input/output histories $IOspace_0$). The definition is satisfied if the above operations on a set of time-invariant input/output histories depends only on the time length (duration) of the histories and not on any specific time interval during which the histories may have been observed.

$$\begin{tikzcd} & IOspace \arrow[dl] \arrow[dr] & \\ \Omega & & \Psi \end{tikzcd}$$

Figure 2.1, Projection of *IOspace* into its first and second coordinates

Henceforth, I shall use *IOspace* instead of $IOspace_0$ for a less cluttered notation. Remember that, algebraically, only the fact that time possesses a zero point is important, not the precise location of that zero point along the time axis.

When a system is time-invariant, its behavior is independent of the selection of the zero time point. Input/output histories that are jointly invariant under translation are consequently called time-invariant input/output histories.

Let me suppose that a system receives an input history $\omega \in < t_i, t_f >$. The system generates the output history $\psi \in < t_i, t_f >$. Let me further suppose that the same system is exposed at some other time to the input trajectory $\omega' = TRANS_\tau(\omega)$. If the system is time-invariant, it will react by generating the output trajectory $\psi' = TRANS_\tau(\psi)$.

Definition 4 *A set of time-invariant input/output histories is defined as:*

$$\begin{aligned} \textit{time-invariant } IOspace = \; & \{(\omega, \psi) \mid (\omega, \psi) \in IOspace, \\ & cut.shift_\tau(\omega) \in \mathrm{PJN}(IOspace, 1), \\ & cut.shift_\tau(\psi) \in \mathrm{PJN}(IOspace, 2), \\ & for\ \tau \in T.\} \end{aligned}$$

where the cut.shift operator is well-defined.

That is, a set of input/output histories is time-invariant iff it is closed under the well-defined operation *cut.shift.*

The concept of time-invariance was here defined for input/output pairs of histories. This may seem unnecessary. However, it makes sense to define a single history as time-invariant if it possesses the property of translation. However, this is a somewhat weaker definition, since the former definition requires in addition that the same translation τ applies to both the input history and the output history.

The property of determinism of a set of input/output histories states that any input history has a unique output history [MT89, Sho88, Wym93, Zei76]. For instance, given an input trajectory ω, there exits a unique output trajectory ψ.

Note that, although it is meaningful to discuss the notion of time-invariance in terms of a single history, the notion of determinism/causality involves at least two histories; one being the *cause* (input history), and the other being the *effect* (output history).

Definition 5 *A functional IOspace, denoted as $IOspace^{func}$, is a set such that:*

$$\begin{aligned} IOspace^{func} = \; & \{(\omega,\psi) \mid (\omega,\psi) \in IOspace, \\ & (\omega,\psi_1) \in IOspace \;\wedge\; (\omega,\psi_2) \in IOspace \implies \psi_1 = \psi_2\}. \end{aligned}$$

Definition 6 *An IOspace that is both functional and closed under the cut.left operation is said to be causal; i.e.,*

$$\begin{aligned} IOspace^{causal} = \; & \{(\omega,\psi) \mid \\ & (\omega,\psi) \in IOspace^{func} \quad \implies \quad (\omega_{t>},\psi_{t>}) \in IOspace^{func}, \\ & for\ t \in dom(\omega)\}. \end{aligned}$$

For example, suppose we have trajectories (ω,ψ), $(\omega',\psi') \in IOspace^{causal}$. Using the above definitions, we have:

$$\omega_{t>} = \omega'_{t>} \qquad \Rightarrow \qquad \psi_{t>} = \psi'_{t>}$$

where $\omega,\omega' \in \Omega$, $\psi,\psi' \in \Psi$, and $t \in dom(\omega) \cap dom(\omega')$. Hence, from the above definitions, we have:

Proposition 2 *A set of causal, time-invariant input/output histories IOspace is:*

- *functional,*
- *closed under the cut.left operation, and*
- *closed under the cut.shift operation.*

Note that the *causality of a system* is usually defined differently. A system is said to be causal if, at no point in time, its outputs depend on future values of its inputs. My definition, of course, implies the more common definition given above, but does it restrict it further? I must extend my definition in two important points to avoid any further restriction of generality.

The first generalization necessary is the following. The definitions of causality and time-invariance, as provided so far in this text, do not depend on the *states* of the system; in fact, I never even talked about systems having internal states. Yet, state variables are evidently an essential facet of any system with internal memory.

Given the system:

$$\dot{z} = -z + x \qquad ; \qquad z(t_i) = z_i$$

obviously, the output trajectory $z(t)$ does not only depend on the input trajectory $x(t)$, but also on the initial state z_i. Thus, its *IOspace* would not be functional according to my previous definition; consequently, this would not be a causal system either. Clearly, this is not meaningful.

One approach to resolve this apparent inconsistency is to augment the input space with the set of initial internal states of the system, i.e., ω now denotes the set of input trajectories and the set of initial states of the system. Since the input/output pairs only represent certain features of a system's behavior, while ignoring other equally pertinent facets, I/O histories would appear to be inconsistent. In order to ensure consistency, I need to augment the representation with the initial states of the system. By associating the states with the input space, the formerly given definitions can be preserved. They only need to be reinterpreted in a more general sense.

The second generalization necessary is the following. Until now, we were always talking about individual input/output pairs. This is insufficient. In order to guarantee consistency of any of the outputs, we need to look at the entire input space at once. In other words, a multi-input/multi-output (MIMO) system with m inputs and p outputs can be decomposed without reduction of generality into a set of p multi-input/single-output (MISO) systems, but not into $m \times p$ single-input/single-output (SISO) systems. This further simplification is only permitted if I can guarantee that the ignored inputs remain *constant* during the investigation.

More Notation: The definitions for $IOspace$, $IOspace^{func}$, and $IOspace^{causal}$ assume the availability of *all* eligible input/output histories. If only a subset of the eligible input/output histories is considered, I prefix the aforementioned definitions with the term *partial.*

2.2 Generator Segments

This section follows closely the treatment presented in [Zei76]. I stated above that Ω_0 is a semigroup. Of course, Ω_0 contains all histories (trajectories and segments) that begin at $t = 0$. However, it would be much more desirable (for computational efficiency) if I could find a smaller set of histories that can be used to generate all members of Ω_0. That is, I would like to find a set $\Omega_G \subset \Omega_0$ (preferably $|\Omega_G| \ll |\Omega|$) such that the semigroup Ω_G^+ generated by Ω_G results in $\Omega_G^+ \supseteq \Omega$. Ω_G^+ denotes the *composition closure* of Ω_G. Simply stated, Ω_G^+ is the set of segments resulting from finite concatenations of all possible segments. If the set Ω_G satisfies $\Omega_G^+ = \Omega$, it is said to be a set of *generators* for Ω.

Note that ultimately I am interested in a finitely describable generator set. Therefore, if $\omega \in \Omega$, and Ω_G is a generator set for Ω, then $\omega = \omega_1 \circ \ldots \circ \omega_n$ and $\omega_1, \ldots, \omega_n \in \Omega_G$; that is, ω can be *decomposed* into $\omega_1 \circ \ldots \circ \omega_n$ given Ω_G. However, in general, even if there exists for each $\omega \in \Omega$ a decomposition into elements of Ω_G, this decomposition may not be unique. For example, $\omega_1, \ldots, \omega_n$ and $\omega_1', \ldots, \omega_m'$ may both be decompositions: $\omega = \omega_1 \circ \ldots \circ \omega_n = \omega_1' \circ \ldots \circ \omega_m'$, and $\omega \in \Omega_G^+$.

This leads me to the question whether there exist any "canonical decompositions" of $\omega \in \Omega$ into elements from Ω_G that have the same desirable properties. A *canonical decomposition* is any decomposition that is uniquely identifiable.

In particular, I may be looking for decompositions into elements from Ω_G with the smallest possible cardinality, i.e., decompositions with the smallest possible number of elements from Ω_G. I call such a decomposition a *minimum cardinality segmentation* (*mcs*). An algorithm determining an *mcs* can be found using a variant of Bellman's *optimality principle*. Let me assume that the cardinality of the *mcs* is n. Thus, I need to find a cut τ such that $cut.left(\omega, \tau) \in \Omega_G$, and $cut.right(\omega, \tau) \in \Omega$ has an *mcs* of cardinality $n - 1$. Unfortunately, the *mcs* has two serious drawbacks. On the one hand, there may be more than one such decomposition, i.e., the *mcs* is not necessarily a canonical decomposition. On the other hand, since an *mcs* can be found by applying the optimality principle, any algorithm determining an *mcs* must be invariably NP-complete. Consequently, such an algorithm can never be very efficient.

Thus, let me look for a canonical decomposition that can be implemented efficiently. A trajectory ω from Ω can be successively decomposed into elements of Ω_G using the following algorithm. Each *cut* decomposes ω into two parts, whereby the left-cut is always from Ω_G, whereas the right-cut may still be from Ω. Using the rule of the longest match, the cut τ_1 is chosen such that $\ell(\omega_{\tau >}) = \ell(cut.left(\omega, \tau_1))$ is maximized. The cutting procedure is then repeated on $\omega_{<\tau}$, such that $\ell(cut.left(\omega_{<\tau}, \tau_2)$ is again maximized, etc. The iteration ends when the right-cut is from Ω_G. This algorithm describes a canonical decomposition of ω into elements of Ω_G. This canonical decomposition is called the *maximum length segmentation*, (*mls*).

A decomposition $\omega_1, \ldots, \omega_n$ is a maximum length segmentation of ω into elements from Ω_G, if, for each $i = 1, \ldots, n$, $\omega' \in \Omega_G$ being a left segment of $\omega_i \circ \ldots \circ \omega_n$ implies that ω' is a left segment of ω_i.

Definition 7 *If Ω_G generates Ω and if for each $\omega \in \Omega$ an mls decomposition of ω into elements from Ω_G exists, then Ω_G is called an admissible set of generators [Zei76].*

Lemma 1 *If ω has an mls decomposition, it is unique [Zei76].*

The uniqueness property of a decomposition has important consequences as we shall see.

To summarize, the set Ω_G *admissibly generates* Ω_G^+ if Ω_G is an admissible set of generators for Ω_G^+ and if, for each $\omega \in \Omega_G^+$, a unique *mls* decomposition of ω into elements from Ω_G exists.

In the following theorem, a set of sufficient conditions is provided that ensures Ω_G to be an admissible set.

Theorem 1 *Sufficient Conditions for Admissibility: If Ω_G satisfies the following conditions, it admissibly generates Ω_G^+ [Zei76]:*

1. *existence of longest segments: $\omega \in \Omega_G^+ \Rightarrow max\ \{t \mid \ \omega_{t>} \in \Omega_G\}$ exists,*
2. *closure under right-cut: $\omega' \in \Omega_G \Rightarrow \omega'_{<t} \in \Omega_G$ for all $t \in dom(\omega')$.*

It is important to make a few observations about an admissible set of generators. Whenever a trajectory has an *mls*, that decomposition is unique. A trajectory that has an *mls* must be time-invariant, since the composition of a trajectory from the generator set depends only on the duration of generator segments without regard to any fixed reference time point. Furthermore, the decomposability of trajectories into elements from Ω_G and the composition of trajectories out of elements from Ω_G is dependent on the structure of the underlying system.

The definitions $IOspace$, $IOspace^{func}$, and $IOspace^{causal}$ were given for histories in general. If the considered histories are limited to segments, and in particular the generator segments (Ω_G and Ψ_G), we have the following definition for the input/output generator segment space:

Definition 8 *A set of input/output segments*

$$\begin{aligned} IOspace_G = \ & \{(\omega_G, \psi_G) \mid (\omega_G, \psi_G) \in IOspace, \\ & \omega_G \in \Omega_G, \\ & \psi_G \in \Psi_G, \\ & IOspace_G \subseteq IOspace, \\ & dom(\omega_G) = dom(\psi_G)\} \end{aligned}$$

where

$\Omega_G = \mathrm{PJN}(IOspace_G, 1)$ *is the input generator segment set, and*
$\Psi_G = \mathrm{PJN}(IOspace_G, 2)$ *is the output generator segment set.*

Corresponding functionality and causality definitions for input/output generator segments (i.e., $IOspace_G^{func}$ and $IOspace_G^{causal}$) can be given using the $IOspace_G$ definition.

Finally I note that my discussion of generator segments so far has not explicitly referenced the states of the system. I say that a *Complete* I/O segment pair (in the case of a single-input system) is associated with an initial state, in order to overcome the lack of distinguishability among apparently inconsistent I/O segments (cf. Section 2.3.3). The term "Complete" is used to indicate that a system's I/O segment augmented by the set of initial states contains enough information to ensure its uniqueness in relation to other I/O segments of the same system. In other words, the interpretation of Complete is that I/O segments are causal.

In my future discussions, causality shall always be assumed when I speak about individual I/O segments, or $IOspace_G$.

2.2.1 Composition of Functions

Given a pair of functions $f : X \rightarrow Y$ and $g : Y \rightarrow Z$, I can construct a function $h : X \rightarrow Z$.

Definition 9 *The composition of a function $g : Y \rightarrow Z$ with a function $f : X \rightarrow Y$, denoted as $g(f(x))$, is the function $g \circ f : X \rightarrow Z$ [Sto73].*

Earlier, I discussed the composition of histories. Now, given the composition of functions, I can define the notion of well-definedness of a composite function that operates on finite multiple *mls* decompositions (e.g., $\omega, \omega' \in \Omega_G^+$).

Suppose we have a function $\delta_G : Q \times \Omega_G \rightarrow Q$ where Ω_G is as defined above and Q is a set. The function $\delta_G(q, \omega)$ operates on single segments $\omega \in \Omega_G$. This function is referred to as a *single segment function* [Zei76].

Definition 10 *A function $\delta_G^* : Q \times \Omega_G^+ \rightarrow Q$ is called the extension of δ_G [Zei76]:*

$$\delta^*_G(q,\omega) = \begin{cases} \delta_G(q,\omega) & \mathit{for} \quad \omega \in \Omega_G, q \in Q \\ \delta^*_G(\delta_G(q,\omega_1),\omega_2,\ldots,\omega_n) & \mathit{for} \quad \omega \in \Omega^+_G, q \in Q \end{cases}$$

where $\omega_1, \ldots, \omega_n$ is the mls decomposition of ω.

The extended function $\delta^*_G(q,\omega)$ is a composition of single segment functions that operate on single segments. That is, I can rewrite $\delta^*_G(q,\omega)$ as:

$$\delta^*_G(q,\omega) = \delta^*_G(q,\omega_1,\ldots,\omega_n) = \delta_G(\delta_G(\ldots\delta_G(q,\omega_1),\omega_2),\ldots),\omega_n)$$

where $\omega_1, \ldots, \omega_n$ is the *mls* decomposition of ω. Since $\delta_G(q,\omega_i) = q'$ for $q' \in Q$ and $\omega_i \in \Omega_G$, I can use the composition of functions as defined above along with the *mls* decomposition for ω to ensure that $\delta^*_G(q,\omega)$ exists and is unique for $q \in Q$ and $\omega \in \Omega^+_G$.

Proposition 3 *A function $\delta^*_G : Q \times \Omega^+_G \rightarrow Q$ is well-defined if it exists (i.e., Ω_G admissibly generates Ω^+_G) and is unique for each $q \in Q, \omega \in \Omega^+_G$ [Zei76].*

For example, consider the situation where δ^*_G operates on two histories ω and ω' with the *mls* decompositions $\omega_1 \circ \ldots \circ \omega_n$ and $\omega'_1 \circ \ldots \circ \omega'_m$, respectively. In particular, if $\delta^*_G(q, \omega \circ \omega'))$ and $\delta^*_G(\delta^*_G(q,\omega),\omega')$ are identical, I say that δ^*_G has the *composition property.* That is, the composition property has to be satisfied by showing that the *mls* decomposition of $\omega \circ \omega'$ is the same as the *mls* decomposition of ω followed by the *mls* decomposition of ω' if δ^*_G is to be well-defined.

2.3 Abstract Stratification of Models

To formally treat various forms of representing a "system," the interface among systems, and recursive decomposability, a stratification of system specifications is shown below [Zei76][3].

Level 0:	Observation Frame (O)
Level 1:	Input/Output Relation Observation (IORO)
Level 2:	Input/Output Function Observation (IOFO)
Level 3:	Input/Output System (IOS)
Level 4:	Structured Input/Output System (SIOS)
Level 5:	Coupling of Systems (CS)

I shall utilize this view of a system stratification throughout this work. As the name of each specification indicates, more concreteness is introduced as I proceed from the lowest level to ever higher levels through the hierarchy. Relationships between certain levels are discussed. In particular, Levels 1, 2, and 3 are considered in detail. The associations between two neighboring levels are established in both directions; that is, we have downward and upward associations. The former is in the direction of structure to behavior, while the latter attempts to hypothesize structure from behavior.

Before proceeding further, I note that I am considering systems without restricting them to any particular formalism, such as the continuous-time, the discrete-time, or the discrete-event formalism. Thus, the nature of the time base and the input/output value sets are unspecified. Correspondingly, the following levels of system specifications are valid for any of the classes of histories discussed earlier.

[3] There are other approaches to classification of systems [Kli85, MT89, Wym93].

2.3.1 Input/Output Observation Frame

The Observation Frame (O) is a structure defined as [Zei76]:

$O = \langle T, X, Y \rangle,$ where:

T	Time Base
X	Input Value Set
Y	Output Value Set

T is the time base, and X and Y are sets containing the legal values that the input and output variables may assume.

2.3.2 Input/Output Relation Observation

The Input/Output Relation Observation (IORO) is a structure [Zei76]:

$IORO = \langle T, X, \Omega, Y, R \rangle$ where:

T	time base
X	input value set
Y	output value set
Ω	input history set
R	I/O relation set

R is a set of Input/Output relations with the following constraints: $\Omega \subseteq (X, T)$, $R \subseteq \Omega \times (Y, T)$. Also, if $(\omega_i, \psi_i) \in R$ for $i = 1, 2, \ldots$, then $dom(\omega_i) = dom(\psi_i)$. The remaining elements of the IORO are already defined in the Observation Frame. The pair (ω, ψ) is called an *input/output* history pair, where $\omega \in \Omega$ and $\psi \in (Y, T)$.

2.3.3 Input/Output Function Observation

The Input/Output Function Observation (IOFO) is a structure [Zei76]:

$IOFO = \langle T, X, \Omega, Y, F \rangle$ where:

T	Time base
X	Input Value Set
Y	Output Value Set
Ω	Input History Set
F	I/O Function Set

F is a set of input/output functions with the constraint that:

$$f \in F \qquad \text{implies} \qquad f \subseteq \Omega \times (Y, T) \qquad \text{is a function.}$$

Thus, F must satisfy $dom(\omega_i) = dom(\psi_i)$ for any $f = \{(\omega_1, \psi_1), \ldots, (\omega_n, \psi_n)\}$ and $\omega_i \in \Omega, \psi_i \in (Y, T)$. The remaining elements of the quinary IOFO were defined previously.

Each $f \in F$ is associated with a unique initial state in order to ensure a unique response, $\psi_i = f(\omega_i)$, for an input segment, ω_i. At this level, we only know the initial state for each f. Each f is a grouping of those Input/Output segments that are associated with a unique initial state. Note how we are associating an initial state with each (ω_i, ψ_i) pair as I alluded to in discussing Complete and Incomplete I/O segments in Section 2.2.

2.3.4 Input/Output System

The Input/Output System (IOS) is a structure [Zei76]:

$IOS = \langle T, X, \Omega, Q, Y, \delta, \lambda \rangle$ where:

T	time base
X	input value set
Y	output value set
Ω	input history set
Q	state set
δ	transition function
λ	output function

The transition function is defined as $\delta : Q \times \Omega \rightarrow Q$ where Q represents the memory of the system. The output function is defined as $\lambda : Q \rightarrow Y$.

The input/output system structure must satisfy the following axioms:

1. Ω is closed under composition,
2. the structure is deterministic in terms of δ and λ, and
3. the composition property of $\delta(q, \omega \circ \omega') = \delta(\delta(q, \omega), \omega')$ is ensured for every pair of contiguous histories $\omega, \omega' \in \Omega$.

2.3.5 Structured Input/Output System

My future discussions do not depend on system specification at this level. However, for the sake of completeness, it is sufficient to say that a system specified at this level differs basically from the IO system in that its sets and functions are structured. In other words, the sets and functions are represented as more elementary sets and functions. For a comprehensive exposition refer to [Zei76].

2.3.6 Coupled System

Just as in the previous case, I refrain from giving a description of a coupled system. It suffices to say that a system represented at this level designates what other (less complex) systems are to be coupled and how they interact with each other. Again, refer to [Zei76] for further information.

2.4 Discrete-Event Specifications

As I stated earlier, models can be represented in at least three separate formalisms: the continuous-time formalism (differential equation formalism), the discrete-time formalism (difference equation formalism, finite automata formalism), and the discrete-event formalism.

[Zei76] chose the I/O system as the basic level for discussing the role of modeling formalisms. The selection of the I/O system as the most appropriate basic level appears to be triggered by an emphasis on deductive modeling approaches.

[Zei76] then formalized the I/O system level within the discrete-event formalism. In the following two sections, I shall present the discrete-event I/O system and the discrete-event I/O function observation specifications.

2.4.1 Discrete-Event I/O System Specification

A discrete-event system specification (*DEVS*) is a structure [Zei76]:

$M = \langle X_M, S_M, Y_M, \delta_M, \lambda_M, t_a \rangle$ where:

X_M is a set of external input events
S_M is a set of sequential states
Y_M is a set of output events
δ_M is the transition function
λ_M is the output function
t_a is the time advance function

The above structure has the following constraints:

1. The total state of the system determines the state of the system. It contains two state variables s and e:

$$Q_M = \{(s, e) \mid s \in S_M, 0 \leq e \leq t_a(s)\}$$

where e is the elapsed time in state s. S_M is called the sequential state to distinguish it from the total state Q_M.

2. The time advance function t_a is defined as:

$$t_a : S_M \longrightarrow \Re^+_{0,\infty}$$

where t_a is interpreted as the maximum time the system can remain in state $s \in S_M$.

3. The transition function is defined as

$$\delta : \; Q_M \times (X_M \cup \{\phi\}) \longrightarrow S_M.$$

δ_M is a shorthand notation to represent two different functions — the external transition function δ_{ext} and the internal transition function δ_{int}. The external transition function is defined as:

$$\delta_{ext} : Q_M \times X_M \longrightarrow S_M.$$

The interpretation of δ_{ext} is that, if an event $x \in X_M$ arrives and the system has been in state $s \in S$ for an elapsed time $0 \leq e \leq t_a(s)$, it will transition to $\delta_{ext}(s, e, x)$ instantaneously. At the same time, the elapsed time is initialized to zero (i.e., $e = 0$).

The internal transition function is defined as:

$$\delta_{int} : S_M \longrightarrow S_M.$$

The internal transition function causes the system to change its state internally after $t_a(s)$ units of time have elapsed if no external event has arrived in between. Simultaneously, the elapsed time component e is initialized to zero just as in the case of the external transition function.

4. The output function is defined as:

$$\lambda : \; Q_M \longrightarrow Y_M.$$

2.4.2 Discrete-Event I/O Function Observation Specification

If the set of input/output histories is causal and time-invariant, then $IOFO = \langle T, X, \Omega, Y, F \rangle$ is a causal and time-invariant specification. The I/O function set F can be represented as:

$$\begin{array}{lll} F = & \{f_i \mid f_i = (s_i, g_i), & \begin{array}{rl} g_i : & \{(\omega_{i,1}, \psi_{i,1}), \cdots, (\omega_{i,n}, \psi_{i,n})\} \\ s_i : & \text{is an initial state associated with } g_i \end{array} \\ & \omega_{i,j} \in \Omega, \;\; \psi_{i,j} \in (Y,T) & \quad \text{for} \quad i = 1, 2, \ldots, \;\; j = 1, \ldots, n\}. \end{array}$$

Hence a reformulation of $IOFO = \langle T, X, S_i, Y, IOspace, F \rangle$, represents the causal, time-invariant IOFO with $IOspace$ being the set of input/output histories (cf. Definition 6). This alternative form of the IOFO specification explicitly represents the initial states S_i.

This IOFO can be further specialized into its discrete-event form. A discrete-event IOFO must operate on discrete-event histories, i.e.,

Definition 11 *A set of discrete-event input/output histories $DEVS(X,Y)$, beginning at time zero, satisfies:*

$$DEVS(X,Y) = \{(\omega, \psi) \mid \omega \in DEVS(X), \quad \psi \in DEVS(Y) \;\; \textit{where}:$$

$$DEVS(X) = \Omega^{+}_{\phi,x} = \bigcup_{i \in I+} \Omega^{i}_{\phi,x} \quad \textit{such that}$$

$$\Omega^{i}_{\phi,x} = \{\omega^{1}_{\phi,x} \circ \ldots \circ \omega^{i}_{\phi,x} \mid \omega^{j}_{\phi,x} \in \Omega_{\phi,x}, \;\; j = 1, \ldots, i, \qquad \Omega_{\phi,x} = \Omega_{\phi} \cup \Omega_{x}\},$$

$$DEVS(Y) = \Psi^{+}_{\phi,y} = \bigcup_{i \in I+} \Psi^{i}_{\phi,y} \qquad \textit{such that}$$

$$\Psi^{i}_{\phi,y} = \{\psi^{1}_{\phi,y} \circ \ldots \circ \psi^{i}_{\phi,y} \mid \psi^{j}_{\phi,y} \in \Psi_{\phi,y}, \;\; j = 1, \ldots, i, \qquad \Psi_{\phi,y} = \Omega_{\phi} \cup \Psi_{y}\}.$$

subject to:

$$\Omega_{\phi} = \{\phi_{\tau} \mid \tau \in \Re, \phi_{\tau} : < 0, \tau > \longrightarrow \{\phi\}\},$$

$$\Omega_{x} = \{x_{\tau} \mid \tau \in \Re, x_{\tau} : < 0, \tau > \longrightarrow X^{\phi},\ x_{\tau}(0) = x,\ x_{\tau}(t) = \phi \quad for \quad t \in (0, \tau)\},$$

$$\Psi_{y} = \{y_{\tau} \mid \tau \in \Re, y_{\tau} : < 0, \tau > \longrightarrow Y^{\phi},\ y_{\tau}(0) = y,\ x_{\tau}(t) = \phi \quad for \quad t \in (0, \tau)\}.$$

The above sets $\Omega_{\phi}, \Omega_{x}$, and Ω_{y} contain discrete-event segments.

Proposition 4 *A set of discrete-event, causal, time-invariant input/output histories, denoted as $IOspace_{D}$, is:*

- *functional,*
- *closed under the cut.left operation, and*
- *closed under the cut.shift operation.*

Given Proposition 4 and the alternative form of the IOFO, I define a discrete-event, causal, time-invariant I/O function observation specification (DEVF) to be a structure:

$$D = \langle X_{D}, S_{i}, Y_{D}, IOspace_{D}, F_{D} \rangle \qquad \text{where:}$$

$X_{D} = \{x \mid x \in \{reals, symbols\}, \quad x \in range(\omega(t)),$
for some finite subset of $t \in dom(\omega)$ and $\omega \in PJN(IOspace_{D}, 1)\}$,

S_{i} : A set of initial states associated with (ω, ψ) such that $(\omega, \psi) \in IOspace_{D}$,

$Y_{D} = \{y \mid y \in \{reals, symbols\}, \quad y \in range(\psi(t)),$
for some finite subset of $t \in dom(\psi)$ and $\omega \in PJN(IOspace_{D}, 2)\}$,

$IOspace_{D} \subseteq DEVS(X_{D}, Y_{D})$: A set of discrete-event, causal, and time-invariant I/O histories,

$F_{D} = \{f_{i} \mid f_{i} = (s_{i}, g_{i}), \quad s_{i}$: is an initial state associated with g_{i},
$g_{i} \subseteq IOspace_{D} \quad$ for $\quad i = 1, 2, \ldots\}$

Causality: For any $f = (s, g) \in F_D$, we have:

$$\omega_{t>} = \omega'_{t>} \quad \Longrightarrow \quad \psi_{t>} = \psi'_{t>}$$

where $(\omega, \psi), (\omega', \psi') \in g$, and $t \in dom(\omega) \cap dom(\omega')$.

Time-invariance:

$$(s, g) \in F_D \quad \Longrightarrow \quad (s, g') \in F_D \qquad \text{where:}$$

$g : \{(\omega_1, \psi_1), \cdots, (\omega_n, \psi_n)\} \subseteq IOspace_D,$
$g' : \{(\omega'_1, \psi'_1), \cdots, (\omega'_n, \psi'_n)\} \subseteq IOspace_D,$
$\omega'_j = cut.shift_\tau(\omega_j)$ for $j = 1, \ldots, n,$
$\psi'_j = cut.shift_\tau(\psi_j)$ for $j = 1, \ldots, n.$

The causality and time-invariance of the IOFO specification are based on knowledge of the initial states associated with input/output histories.

Chapter 3 An Inductive Modeling Approach

One of the essential steps in developing a useful approach to solving a problem is to figure out how it should be perceived. For instance, as we shall see in the following chapter, there have been intense efforts to find an appropriate representation that would support common-sense reasoning adequately. In such cases, representation of a problem in a way that is amenable to solving it, then becomes indispensable. Thus, the way a problem may be approached and eventually solved often hinges upon its representation. We may seek different representations of a problem for different purposes. For example, we may want to gain a better understanding of the problem at hand, thus looking for representations that would best support this effort.

In this chapter, I look first at a different representation of the IO system that will help us see some of its in-depth characteristics. Then, I introduce a new IOFO structure — called iterative IOFO structure — to have a of representing input/output segments instead of input/output trajectories.

3.1 Iterative Specification of I/O System

In this section, I shall discuss the so-called *iterative specification* of I/O systems [Zei76]. The approach to derive the iterative I/O system specification is a general one in the sense that it is applicable to any I/O system regardless of the system formalism it might be expressed in.

In the previous chapter, I looked at the structure of the discrete-event I/O system, $M = \langle X_M, S_M, Y_M, \delta_M, \lambda_M, t_a \rangle$. This structure specifies an I/O system in a compact way. Whereas it does provide the essential information that is necessary for its unique specification, it cannot be used to describe some of the fundamental properties and features of an I/O system.

Hence the motivation for iterative specification is to provide a means to delve into the subtleties of the fundamental properties of the I/O system structure. For instance, the nature of the input set Ω and how the transition function δ_M operates can be described succinctly once the iterative I/O system specification is available. Having the ability to discuss the elements of the I/O system based on its iterative specification and using the discrete-event formalism, the iterative specification can render the *legitimacy* property of discrete-event systems. A brief account of the legitimacy property will be provided in this chapter.

Since we are interested in understanding the details and properties of I/O systems, it is necessary to begin with I/O segments and develop appropriate functions that can operate on them. The previously discussed generator segments, as well as the functions that operate on them, form the basis that leads to the iterative I/O system specification.

In Chapter 1, I stated that a system can be described using differential equations, automata, or discrete-event formalisms. It is helpful to choose one of these formalisms in order to elucidate the iterative specification exposition. I already confined myself to discrete-event systems, and hence I shall use the discrete-event I/O system for the development of the iterative I/O specification.

It is necessary to know what kind of knowledge is necessary to derive the iterative discrete-event I/O system specification from the discrete-event I/O system structure. As the construction of the iterative I/O system specification is based on generator segments and their corresponding functions, the knowledge we are seeking must come from them.

Given the discrete-event I/O system structure, we know that the time base is real-valued and the input/output histories are of the discrete-event type. This kind of knowledge is referred to as *background conventions* [Zei76], and is common to all types of systems specified (e.g., I/O function and I/O system) within the discrete-event formalism.

Thus, the goal is to take an I/O system M and transform it into its iterative form by selecting the background conventions and by using generator segments and their corresponding functions. A full account of the iterative I/O system specification is presented in [Zei76].

The previously mentioned transformation is a precursor to another transformation. We can use another kind of knowledge to distinguish between members of the same class of systems. For instance, this knowledge dictates how the transition function of a discrete-event I/O system should be extended to operate on composite segments or trajectories. This kind of knowledge is encoded in the *specificity rules* [Zei76]. Such rules specify how a particular function which is defined at the primitive level (i.e., iterative specification) can be extended to be applicable at the extended iterative I/O system level. That is, given a proper definition for a function that operates on a single segment, the correctness of an extended form of the same transition function operating on a history (a set of contiguous segments) would be warranted by the specificity rules.

Then, given an iterative I/O system specification, denoted as G_S, as well as the specificity rules, the extended iterative I/O system, denoted as $IOS(G_S)$, can be derived. Thus, these two transformations systematically generate $G_S(M)$ and $IOS(G_S(M))$ from a discrete-event I/O system M (cf. Figure 3.1 [Zei76].)

If the transformations are *correct*, then M and $IOS(G_S(M))$ should exhibit identical behaviors to the extent that the behavior of M is entailed by that of $IOS(G_S(M))$.

Now we can observe that the underlying purpose for an iterative specification

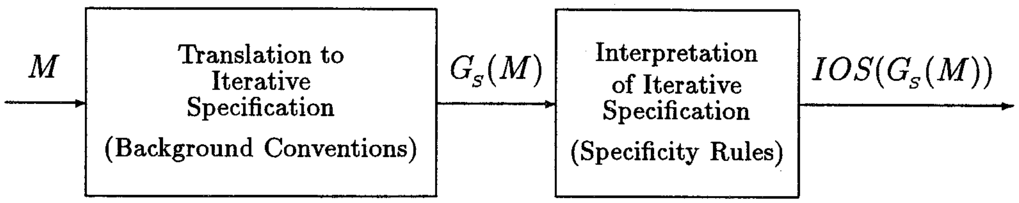

M
Translation to Iterative Specification
(Background Conventions)
$G_S(M)$
Interpretation of Iterative Specification
(Specificity Rules)
$IOS(G_S(M))$

Figure 3.1, Translation and interpretation of I/O system iterative specification

could also be to show the correctness of the higher level I/O system specification by establishing its correctness at the level of its primitive iterative I/O system specification.

In the following two sections, I present the main two theorems developed in [Zei76]. One establishes the conditions that must be satisfied to ensure that an iterative I/O system, when extended, results in an I/O system. The second theorem states that the extended iterative I/O system $IOS(G_S)$ derived from a legitimate discrete-event I/O system is a time-invariant I/O system.

3.1.1 Iterative I/O System Structure

An iterative specification for an I/O system is a structure [Zei76]:

$G_S = \langle T, X, \Omega_G, Q, Y, \delta_G, \lambda \rangle$ where:

T	time base set
X	input value set
Ω_G	input segment generator set
Q	state set
Y	output value set
δ_G	single segment transition function.
λ	output function.

with the following constraints:

$\Omega_G \subseteq (T, X)$, Ω_G is an admissible set of generators,

$\delta_G : Q \times \Omega_G \longrightarrow Q$,

$\lambda : Q \longrightarrow Y$, and

$\delta_G^* : Q \times \Omega_G^+ \longrightarrow Q$ has the composition property.

where as I mentioned earlier, in my notation, I am using Ω in place of Ω_0. We have the following theorem.

Theorem 2 *Let $G_S = \langle T, X, \Omega_G, Q, Y, \delta_G, \lambda\rangle$ be a structure with T a time base set; X, Q, Y the input, state, and output sets, respectively; $\Omega_G \subseteq (T, X), \delta_G : Q \times \Omega_G \longrightarrow Q$; and $\lambda : Q \longrightarrow Y$. Then, if the following conditions hold, G_S is an iterative specification and $IOS(G_S)$ is an I/O system [Zei76].*

1. *Existence of the longest segments:*

   $\omega \in \Omega_G^+ \Longrightarrow max\{t \mid \omega_{t>} \in \Omega_G\}$ *exists,*

2. *Closure under right segmentation:*

   $\omega \in \Omega_G \Longrightarrow \omega_{<t} \in \Omega_G$ *for* $t \in dom(\omega)$,

3. *Closure under left segmentation:*

   $\omega \in \Omega_G \Longrightarrow \omega_{t>} \in \Omega_G$ *for* $t \in dom(\omega)$,

4. *Consistency of composition:*

   $$\omega_1, \ldots, \omega_n \in \Omega_G \;\wedge\; \omega_1 \circ \ldots \circ \omega_n \in \Omega_G^+ \Longrightarrow$$
   $$\delta_G(\omega_1 \circ \ldots \circ \omega_n) = \delta_G((\ldots \delta_G(q, \omega_1), \ldots), \omega_n).$$

The proof of the theorem shows that $\delta_G^* : Q \times \Omega_G^+ \rightarrow Q$ has the composition property.

Now that we have a definition for the iterative I/O system structure, I can briefly say what it means for a discrete-event system to be legitimate.

In simple terms, the legitimacy of the DEVS M ensures that time continues to advance as long as there is some state change. Put in other words, there cannot be a sequence of states that would prevent time to advance beyond a certain point. I stated earlier that the time advance function should be discussed using an iterative I/O system specification. The following formulation of DEVS M legitimacy shows why.

If we let $\sum(s,n)$ be the total time necessary to make n transitions beginning in state s, then we say DEVS M is legitimate if for each $s \in S$, $\lim_{n\rightarrow\infty} \sum(s,n) \rightarrow \infty$ [Zei76].

Note that each state change is due to a single segment function δ_G acting on a generator segment $\omega \in \Omega_G$, hence the importance of the iterative specification in defining the notion of legitimacy.

Earlier we saw a definition for G_S. Using the specificity rules pertaining to Ω^+_G and δ^*_G in conjunction with $G_S = \langle T, X, \Omega_G, Q, Y, \delta_G, \lambda\rangle$, we can construct $IOS(G_S)$ as:

$$IOS(G_S) = \langle T, X, \Omega^+_G, Q, Y, \delta^*_G, \lambda\rangle$$

T	time base set
X	input value set
Ω^+_G	extended generator input set
Q	state set
Y	output value set
δ^*_G	extended single segment transition function.
λ	output function.

with the following constraints:

$$\Omega^+_G = \{\omega_{x_1} \circ \ldots \circ \omega_{x_n} \mid x_i \in X^+, \qquad i = 1, \ldots, n; \qquad n = 1, 2, \ldots..\}$$

$\delta^*_G : Q \times \Omega^+_G \longrightarrow Q$, i.e.,

$$\delta^*_G(q,\omega) = \begin{cases} \delta_G(q,\omega) & \text{for} \quad \omega \in \Omega_G, q \in Q \\ \\ \delta^*_G(\delta_G(q,\omega_1),\omega_2,\ldots,\omega_n) & \text{for} \quad \omega \in \Omega^+_G, q \in Q \end{cases}$$

where the *mls* decomposition of ω is $\omega_1,\ldots,\omega_n \in \Omega_G$,

$\lambda \longrightarrow Y$.

3.1.2 DEVS Iterative Specification

Having defined $IOS(G_S)$, we can use it to construct a discrete-event extended iterative specification $IOS(G_S(M))$. We thus have the following main theorem:

Theorem 3 *Let $M = \langle X_M, S_M, Y_M, \delta_M, \lambda_M, t_a \rangle$ be a a legitimate discrete-event I/O system. Then, $G_S(M) = \langle T, X, \Omega_G, Q, Y, \delta_G, \lambda \rangle$ is its iterative specification and $IOS(G_S(M)) = \langle T, X, \Omega^+_G, Q, Y, \delta^*_G, \lambda \rangle$ is a time-invariant I/O system [Zei76].*

Suppose $\tilde{B}$ stands for the collection of all behavior exhibited by the discrete-event I/O system M, and correspondingly, $\tilde{B}'$ denotes the collection of all behavior exhibited by its I/O system iterative specification $IOS(G_S(M))$. Then, we have:

$$\tilde{B}' \supseteq \tilde{B}.$$

3.2 Inductive Modeling and IOFO Specification

In Chapter 1, the inductive modeling paradigm was defined as a modeling paradigm that considers a system as a *black box*, of which nothing is known but its inputs and outputs and attempts to predict a system's behavior based solely on this knowledge. In particular, no knowledge is claimed about the system's internal structure. For example, for a discrete-event I/O system, there must exist explicit knowledge about how states of a system can change (i.e., internal and external transition functions should be defined). As we observed, at the IOFO level, no such transition functions are specified. Therefore, the I/O function system specification provides a suitable mechanism for an inductive system-theoretic approach to modeling.

The iterative specification of I/O systems presented us with insight into its inner structure. My motivation was primarily to show that, to gain a thorough understanding of an I/O system and its intricate properties, it is necessary to study its inner structure. The basic notion of the iterative specification not only can be applied at the I/O system level, but also at the I/O relation observation and the I/O function observation levels.

This observation led me to seek an iterative specification of the IOFO structure. However, my motivation is different from that stated for an I/O system. Instead, my goal is to develop an iterative representation for the IOFO structure. Thereafter, based upon its structure, I can establish a mechanism that allows me to *operate* on it.

Correspondingly, I shall formulate an iterative I/O function observation structure. In doing so, as in the case of the iterative I/O system structure, I shall be able to explore the extent to which it is applicable.

The IOFO specification $\langle T, X, S_i, Y, IOspace, F \rangle$, presented in Section 2.4.2, is assumed to contain knowledge about the initial states S_i that are associated with the respective input/output trajectories $IOspace$. This is one of the main differences between the input/output system level and the input/output function observation level. Consequently, if we have a finite number of input/output trajectories and their initial states, all that we can expect is to match an initial state and its corresponding input trajectory with the repertoire of available data. That is, suppose we have a set F where $f = (s_i, \{(\omega_{i,j}, \psi_{i,j})\}) \in F$ for $i = 1, \ldots, n$, and $j = 1, \ldots, m$. Also, assume we have an initial state s' and input trajectory ω' for which we would like to determine the output trajectory. Then, if there exists f such that one of its input trajectories matches ω' and also its initial state matches s', then the output trajectory ψ is trivially obtained from f.

Obviously, the success of finding a match for a given initial state s' and an input trajectory ω', given F, depends on the richness of F. However, if it contains input/output segments and their final states, then it is possible to not only reconstruct previously observed input/output histories, but also a subset of unobserved input/output histories that can be constructed by concatenating appropriate I/O segments. Hence, although these I/O trajectories have not been observed explicitly, they can be constructed by rearranging some of the available I/O segments.

Additionally, having input/output segments supports hypothesizing those I/O trajectories that cannot be constructed with the existing I/O segments. The IOFO structure cannot support construction of these types of I/O histories. Hence we are interested in defining an *iterative IOFO structure* from the IOFO structure.

To this end, we are led to seek an input/output *segment generator* function, F_G that is similar in structure to F. The elements of F_G have the form $(s, \{(\omega_G, \psi_G)\})$ where $\omega_G \in PJN(IOspace_G, 1), \psi_G \in PJN(IOspace_G, 2)$. Constructing F_G requires partitioning of observed I/O trajectories (ω, ρ) into segments $(\{\omega_{\nless,\ngtr}\}, \{\psi_{\nless,\ngtr}\})$.

We may envision two stages in generating F_G, starting from a set of observed input/output histories F. In the first stage, the *partitioning* of trajectories into segments based on the segments' granularity, duration, and initial state assignment is dependent on a set of conjectures (assumptions). These assumptions constitute some belief set, which I denote as *assumption set-I.* This belief set, therefore, forms the basis upon which an iterative IOFO specification can be formalized.

In partitioning a trajectory into segments, if we consider two contiguous I/O segment pairs, we are assuming that the initial state of one pair immediately following another pair is the same as the final state of the previous pair. This is necessary since the systems that we are considering are assumed to be causal, and we are merely partitioning I/O trajectories.

If we are to *compose* trajectories from segments, another belief set is also needed. I denote this second belief set as *assumption set-II.* To compose two segments ω_1 and ω_2, they may be concatenated as $\omega_1 \circ \omega_2$. However, unless the final state of ω_1 is the same as the initial state of the ω_1, we are forced to make $\omega_1 \circ \omega_2$ a hypothesis. That is, ignoring a mismatch between the final state of ω_1 and the initial state of the ω_2 results in a hypothesized trajectory. This explains the requirement for assumption set-II. Later, in chapter 5, I shall extend this definition of the assumption set-II.

At the IOFO specification level, our two transformations correspond to two sets of assumptions I and II. As motivated above, the former set facilitates the *partitioning* of trajectories into segments resulting in the *iterative IOFO structure.* The latter set allows the composition of incompatible I/O segments. Thereby construction of composite segments (trajectories) from segments leads to the *free iterative IOFO structure.* I denote the iterative IOFO specification as G_F and the free iterative IOFO specification as $IOFO(G_F)$. Hence,

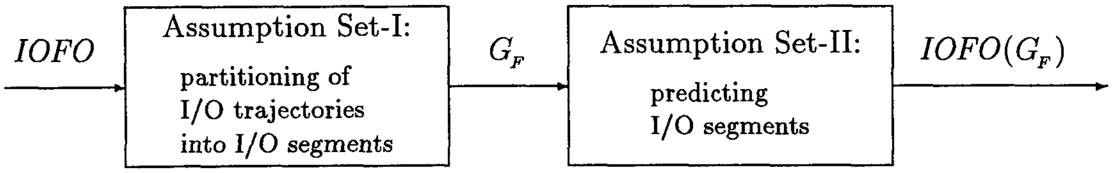


Figure 3.2, Role of assumptions in constructing G_F and $IOFO(G_F)$.

$$IOFO \longrightarrow G_F \longrightarrow IOFO(G_F).$$

Figure 3.2 depicts how each assumption set contributes to arriving at G_F and $IOFO(G_F)$.

The transformations from $IOFO$ to G_F and from G_F to $IOFO(G_F)$ can be considered to be independent. The interpretation of the independence of these transformations is that, once G_F is formed, the construction of $IOFO(G_F)$ is solely dependent on the assumption set-I.

There is, nevertheless, some interdependence between the transformations. This is due to the fact that, in constructing composite I/O trajectories, some segments may be hypothesized. The assumptions underlying the assignment of states of I/O segments for G_F and states of hypothesized I/O segments for $IOFO(G_F)$ might depend on each other. That is, the assumptions underlying the partitioning of a trajectory and the composition of I/O segments can be quite different.

First, I shall focus my attention on how one might partition an I/O trajectory given its initial state into I/O segments with assigned initial states.

Let us assume we have a function γ_G, which we call a *quasi-state prediction function.* This function hypothesizes about a segment's final state based on assumption set-I. Note that the final state of a segment of a trajectory is the same as the initial state of the segment following it. Conversely, the initial state of one segment is the final state of the segment preceding it. In order to identify possible assumptions, it is useful to determine what can be used to characterize γ_G.

In an inductive modeling approach introduced by Klir, during the so-called *recoding process*, the range of a variable is divided into levels or regions [Kli85, Cel91]. For instance, we may associate the symbols 'too low,' 'low,' 'normal,' 'high,' and 'too high' with levels representing regions of systolic blood pressure [Cel91]. Then, given a set of input and output variables, the number of discrete states associated with these variables is defined as:

$$num_{States} = num.levels_{input} \quad * \quad num.levels_{output}.$$

Thence, the number of states for a given system is based on the number of levels assigned to each variable. These states are time independent in the sense that they are not determined based on any particular instant of time.

Given we have the knowledge of I/O segments, the final state of an I/O segment can be one of the following mappings using the elements *time*, *inputs*, and *outputs*:

1. $inputs \longrightarrow states$
2. $outputs \longrightarrow states$
3. $time \longrightarrow states$
4. $inputs \times time \longrightarrow states$
5. $outputs \times time \longrightarrow states$
6. $inputs \times outputs \times time \longrightarrow states$

Thus, the the quasi-state prediction function is:

$$Z \longrightarrow \{S_f \mid \begin{array}{l} Z = \mathcal{P}\{inputs,\ outputs,\ time\} \\ S_f : \text{set of candidate final states.}\} \end{array}$$

where $\mathcal{P}\{inputs,\ outputs,\ time\}$ denotes the power set on $inputs, outputs$, and $states$.

For example, we can use $outputs \longrightarrow candidate.final.states.$ The inclusion of set *time* in the mapping functions provides us with the ability to examine the past of the I/O histories, thus allowing a special class of final states (finite memory machines) to be hypothesized. For instance, we can examine one or several previous segments in order to conjecture what the *state* might be. The inclusion of *time* in the state identification can play a critical role as compared to the *inputs* and *outputs* elements. In other words, if we use, for instance, $outputs \times time \longrightarrow states$, we are allowing the examination of not an output value, but an output trajectory (or any of its parts).

We may represent $inputs \times outputs \times time$ as $R \subseteq IOspace.$ Thus, the quasi-state identification function would be:

$$R \longrightarrow S_f.$$

If we also include the initial state set S_i in the previous mapping, we can formalize the quasi-state identification function for trajectories as:

$$\gamma : S_i \times R \longrightarrow S_f$$

where S_i is the set of initial states identified with R. If we consider segments ($R_G \subseteq IOspace_G$) instead of trajectories, we have:

$$R_G \longrightarrow S_f.$$

Furthermore if we also include the initial state set S_i, as we did for trajectories, with R_G, then the previous quasi-state identification function for segments becomes:

$$\gamma_G : S_i \times R_G \longrightarrow S_f.$$

These mappings are not necessarily the truly representative set of states of the system, but they may be the best that we are able to obtain given the input/output segments and their initial states only. I note that higher level knowledge such as the sets X and Y (elements of IOFO) can lead to different sets of final states. That is, depending on how the number of levels (i.e., the quantization) is specified, the candidate final states may vary.

It might be tempting to form $S_i \cup S_f$; after all, any state can belong to the set of initial states, the candidate final states, or both. We may like to reformulate the quasi-state identification function for the segments as:

$$\gamma_G : S_i \times R_G \longrightarrow S_i.$$

In this formulation the initial state set and the final state set are identical. Given two segments that are to be composed, this form of γ_G is proper, since a segment's final state can be used as the initial state of another. In general, however, we should be cautious and keep these two sets separate; the semantics associated with the initial states and final states can be quite different. The reason is that when it becomes necessary to predict an I/O segment, (e.g., assuming its initial state using assumption set-II), the assumed initial state must not necessarily be valid. That is, although the assumed initial state belongs to S_i, the revised initial state may not.

A state could be part of either an observed or a hypothesized I/O segment. Hence, in general, we do not want to extend S_i to include S_f, or vice-versa. We like to treat initial states associated with hypothesized I/O histories in a manner that allows us to revise our belief in them whenever this should become necessary. In other words, when there is evidence to falsify hypothesized candidate initial states, we can readily do so by replacing them with new ones. The need to maintain two sets of states is necessary when we are interested in the interaction of assumptions between both stages. Nevertheless, in this work, I shall use the latter formulation of γ_G as we deal with assumption set-II alone.

3.2.1 Iterative I/O Function Observation Structure

Using the alternative representation of an IOFO structure given above, I define an iterative specification for a causal, time-invariant I/O function observation structure as:

$$G_F(\gamma_G) = \langle T, X, S_i, Y, IOspace_G, F_G, \gamma_G \rangle \qquad where:$$

T	time base
X	input value set
Y	output value set
S_i	set of initial states
$IOspace_G$	causal, time-invariant input/output segment generator set
F_G	I/O function generator set
γ_G	quasi-state identification function

with the following constraints:

$$F_G : S_i \longrightarrow \text{partial } IOspace_G,$$
$$\gamma_G : S_i \times IOspace_G \longrightarrow S_i.$$

The interpretation of the function γ_G is that it maps a given initial state and an input/output segment pair into another initial state. We can equate (interpret) the final state s_f of $(s_i, (\omega, \psi))$ with the initial state of another pair of input/output segments (ω', ψ') as desired. The quasi-state identification function γ_G as defined above has the most general form in terms of utilizing states, outputs, inputs, and time. For instance a less general form of γ_G would be:

$$\gamma_G : \Psi_G \longrightarrow S_i.$$

To obtain an output segment $\psi_G^{i,j}$ for an input segment $\omega_G^{i,j} \in \Omega_G$, an initial state $s_i \in S_i$, and a given data set F_G (cf. Section 2.4.2), we can define the output segment matching function as:

$$\eta_G : S_i \times \Omega_G \times F_G \longrightarrow \Psi_G,$$

and its *computational* counterpart as:

$$\eta_G(s_i, \omega_G, F_G) = F_G(s_i, \omega_G).$$

The interpretation of the above function is that there exists an $f_i \in F_G$ such that $\eta_G : (s_i, \omega_G^{i,j}, f_i) \mapsto \psi_G^{i,j}$. For example, using the output segment matching function η_G and the special form of the quasi-state identification function $\gamma_G : \Psi_G \rightarrow S_f$, we can determine the final state by consecutive application of γ_G followed by η_G. That is,

$$\gamma_G(\eta_G(s_i, \omega_{i,j}, f_i)) = \gamma_G(\psi_{i,j}) = s_f,$$

or using the general form of the quasi-state identification function, $\gamma_G : S_i \times IOspace_G^+ \rightarrow S_i$, we have the general form for combining η_G and γ_G:

$$\gamma_G(s_i, (\omega_{i,j}, \eta_G(s_i, \omega_{i,j}, f_i))) = \gamma_G(s_i, (\omega_{i,j}, \psi_{i,j})) = s_f.$$

The new iterative IO function observation structure results add one level to the abstract stratification of Section 2.3. That is, we have the following as the extended hierarchy of system specifications.

Level 0:	Observation Frame (O)
Level 1:	Input/Output Relation Observation (IORO)
Level 2:	Input/Output Function Observation (IOFO)
Level 2.5:	Iterative Input/Output Function Observation (G_F)
Level 3:	Input/Output System (IOS)
Level 4:	Structured Input/Output System (SIOS)
Level 5:	Coupling of Systems (CS)

3.2.2 Free Iterative I/O Function Observation Structure

Having defined the iterative IOFO structure *IOFO*, we give the structure of the free iterative IOFO structure $IOFO(G_F)$ generated from G_F:

$$IOFO(G_F) = \langle T, X, S_i, Y, IOspace_G^+, F_G^+ \rangle$$

where $IOspace_G^+$ is the free semigroup generated from $IOspace_G$, and F_G^+ is the free semigroup constructed for each $s \in S_i$:

$$\overline{g}_i^1 = F_G(s_i)$$

$$\begin{aligned}\overline{g}_i^{n+1} = \; & \{(\omega \circ \omega_G, \psi \circ \psi_G) \mid \\ & (\omega, \psi) \in \overline{g}_i^n, \qquad (\omega_G, \psi_G) \in F_G(\gamma_G(s_i, (\omega, \psi))) \\ & (\omega, \psi) \in IOspace_G^+\},\end{aligned}$$

$$F_G^+ : S_i \longrightarrow \text{ partial } IOspace_G^+,$$

$$F_G^+ = \bigcup\nolimits_{j \in J^+} \overline{g}_i^j.$$

The input/output function generator set has the form:

$$\begin{array}{lll} F_G^+ = \{\overline{f}_i \mid \overline{f}_i = (s_i, \overline{g}_i), & \overline{g}_i : & \{(\overline{\omega}_{i,j}, \overline{\psi}_{i,j})\} \\ & s_i : & \text{is an initial state associated with } \overline{g}_i \\ \text{for} \quad i = 1, 2, \ldots, & j = 1, \ldots, n & \\ \overline{\omega}_j = \omega_1 \circ \ldots \circ \omega_m, & \overline{\psi}_j = \psi_1 \circ \ldots \circ \psi_m, & \\ \omega_1, \ldots, \omega_m \in \Omega_G, & \text{and} & \psi_1, \ldots, \psi_m \in \Psi_G\}. \end{array}$$

In order to form $IOFO(G_F)$ from G_F, we need to use the composition of output generator segments, using an extended form of η_G. For example, let us assume $\gamma_G : \Psi_G \rightarrow S_f$, which is a special case of γ_G defined earlier. Also, suppose we have s_i, and two input segments $\omega_{i,j}, \omega_{k,\ell} \in \Omega_G$. The segment $\omega_{i,j}$ is followed by $\omega_{k,\ell}$, and the initial state s_i is associated with $\omega_{i,j}$. Then the following holds:

$$\begin{array}{l} (\omega_{i,j} \circ \omega_{k,\ell}, \psi_{i,j} \circ \psi_{k,\ell}) \in \overline{g}_i, \\ (\omega_{i,j}, \psi_{i,j}) \in F_G(s_i), \\ (\omega_{k,\ell}, \psi_{k,\ell}) \in F_G(s_k), \\ f_i : s_i \mapsto (\omega_{i,j}, \psi_{i,j}) \in F_G, \\ f_k : s_k \mapsto (\omega_{k,\ell}, \psi_{k,\ell}) \in F_G, \text{ and} \\ \eta_G^*(s_i, \omega_{i,j} \circ \omega_{k,\ell}, f_i, f_k) = \psi_{i,j} \circ \psi_{k,\ell}. \end{array}$$

In the above composition of $\psi_{i,j} \circ \psi_{k,\ell}$, the final state of $(\omega_{i,j}, \psi_{i,j})$ is the same as the initial state of $(\omega_{k,\ell}, \psi_{k,\ell})$. If a composite output segment $\psi_{i,j} \circ \psi_{k,\ell}$ can be generated by applying η_G and γ_G on individual segments, then we have:

$$\eta_G(\gamma_G(\eta_G(s_i, \omega_{i,j}, f_i)), \omega_{k,\ell}, f_k) = \psi_{i,j} \circ \psi_{k,\ell}.$$

What this says is that if $\gamma_G(\eta_G(s_i, \omega_{i,j}, f_i)) = \gamma_G(\psi_{i,j}) = s_k'$ and the initial state associated with $\omega_{k,\ell}$ is s_k, then $s_k' = s_k$ must be satisfied in order for η_G^* to hold. However, if we assign (via assumption set-II) s_k'' to be the initial state of $\omega_{k,\ell}$ such that $s_k'' = s_k$, then we are no longer working within the IOFO formalism; we have stepped outside of it by allowing ourselves to have the composition of two I/O segments, one of which does not belong to $IOspace_G$.

Now, I can define the extended form of the output segment matching function as:

$$\eta_G^* : S_i \times \Omega_G^+ \times F_G^+ \longrightarrow \Psi_G^+, \qquad \text{or}$$

$$\eta_G^*(s_i, \omega_{i,j} \circ \omega_{k,\ell} \circ \ldots \circ \omega_{n,q}, f_i, f_k, \ldots, f_n) =$$

$$\eta_G((\ldots \gamma_G(\eta_G(\gamma_G(\eta_G(s_i, \omega_{i,j}, f_i)), \omega_{k,\ell}, f_k)), \ldots), \omega_{n,q}, f_n)$$

Thence, in composing two I/O segments, the final state associated with an I/O segment pair must be the same as the initial state of the I/O segment following it if the matching function η_G^* is to be applicable. That is, η_G^* must use γ_G in concert with each segment $\omega \in \Omega_G$ and its initial state s_i.

Proposition 5 *Given a causal, time-invariant input/output function observation specification for each $s \in S_i$ and $(\omega, \psi) \in IOspace$*

$$\begin{array}{ll} (s, (\omega, \psi)) \in F_G^+ & \textit{iff} \\ \omega = \omega_1 \circ \ldots \circ \omega_m \in \Omega_G^+, & \\ \psi = \psi_1 \circ \ldots \circ \psi_m \in \Psi_G^+, & \\ (\omega_i, \psi_i) \in IOspace_G, & \textit{for} \quad i = 1, \ldots, m. \end{array}$$

Now we may ask the question: when does an IOFO have an iterative specification? If a causal time-invariant input/output function observation specification, $IOFO = \langle T, X, S_i, Y, IOspace, F \rangle$, has an iterative specification, $G_F(\gamma_G) = \langle T, X, S_i, Y, IOspace_G, F_G, \gamma_G \rangle$, then we have:

$$(s,(\omega,\psi)) \in F \qquad \Longrightarrow \qquad (s,(\omega_1 \circ \ldots \circ \omega_m, \psi_1 \circ \ldots \circ \psi_m)) \in F_G^+$$

where s is the initial state associated with (ω_1, ψ_1), $\ell(\omega_k) = \ell(\psi_k)$, for $k = 1, \ldots, m$. Equivalently, we have:

$$F_G^+ \supseteq F.$$

3.2.3 Iterative Discrete-event IOFO Specification

I am interested in knowing how, if we have a time-invariant discrete-event input/output function observation specification, we can find its iterative counterpart. My aim is to be able to represent and operate on I/O segments instead of I/O trajectories as discussed earlier. Note that the I/O trajectories and their initial states are in F. Hence I like to obtain their I/O generator segments and their initial states as F_G using assumption set-I.

The iterative input/output function observation structure G_F was defined without restricting it to any particular time base such as discrete-time. Here, we like to construct an iterative Discrete-event IOFO specification (DEVF).

Proposition 6 *Let* $D = \langle X_D, S_i, Y_D, IOspace_D, F_D \rangle$ *be a causal time-invariant discrete-event input/output function specification. Then, its iterative specification,* $G_F(D) = \langle T, X, S_i, Y, IOspace_G, F_G, \gamma_G \rangle$*, can be constructed provided that:*

1. *The mls decomposition of I/O trajectories* $\{(\omega,\psi) \mid \omega \in PJN(IOspace_D, 1)$ *and* $\psi \in PJN(IOspace_D, 2)\}$ *exists,*
2. *There exists a quasi-state identification function such that* $\gamma_G(s,(\omega_G,\psi_G)) = s_f \in S_i$*,*
3. *The composition of segments, using* η_G *as defined for the iterative I/O function observation structure, is satisfied within the discrete-event I/O function observation representation.*

If the above conditions are satisfied, the constructed discrete-event iterative I/O function observation has the following form:

$$G_F(D) = \langle T, X, S_i, Y, IOspace_G, F_G, \gamma_G \rangle \qquad \text{where:}$$

$$\begin{aligned}
&T = \Re^+_0 \\
&X = X^\phi_{D,mls} \\
&Y = Y^\phi_{D,mls} \\
&IOspace_G \subseteq DEVS_G(X,Y) \\
&F_G : S_i \longrightarrow \text{ causal time-invariant partial } IOspace_G \\
&\gamma_G : S_i \times IOspace_G \longrightarrow S_i,
\end{aligned}$$

where:

$X^\phi_{D,mls} = X_D \cup X_{mls} \cup \{\phi\}$,

X_{mls} is the set of input events obtained from segments generated from the mls decomposition of $PJN(IOspace_G, 1)$,

$Y^\phi_{D,mls} = Y_D \cup Y_{mls} \cup \{\phi\}$

Y_{mls} is the set of output events obtained from segments generated from the mls decomposition of $PJN(IOspace_G, 2)$,

$$\begin{aligned}
DEVS_G(X,Y) = \;& \{(\omega_G, \psi_G) \mid \omega_G \in DEVS_G(X), \psi_G \in DEVS_G(Y) \\
& DEVS_G(X) = \{\omega_G \mid \omega_G \in \Omega_G, \Omega_G = \Omega_x \cup \Omega_\phi\} \\
& DEVS_G(Y) = \{\psi_G \mid \psi_G \in \Psi_G, \Psi_G = \Psi_y \cup \Omega_\phi\}\},
\end{aligned}$$

$\Omega_\phi = \{\phi_\tau \mid \tau \in \Re, \phi_\tau :< 0, \tau > \longrightarrow \{\phi\}\}$,

$\Omega_x = \{x_\tau \mid \tau \in \Re, x_\tau :< 0, \tau > \longrightarrow X^\phi, x_\tau(0) = x, x_\tau(t) = \phi \text{ for } t \in (0, \tau)\}$,

$\Omega_y = \{y_\tau \mid \tau \in \Re, y_\tau :< 0, \tau > \longrightarrow Y^\phi, y_\tau(0) = y, x_\tau(t) = \phi \text{ for } t \in (0, \tau)\}$.

As can be seen, Ω_{ϕ}, Ω_{x}, and Ω_{y} belong to discrete-event type segments. Having constructed the iterative I/O function specification, we can proceed to construct the iterative I/O system specification based on assumption set-II; i.e.,

$$IOFO(G_{F}(D)) = \langle X, \Omega_{G}^{+}, S_{i}, Y, \Psi_{G}^{+}, F_{G}^{+} \rangle$$

Therefore, the behavior of D, a causal time-invariant discrete-event input/output function observation, is exhibited by the behavior of $IOFO(G_{F}(D)$, the free specification constructed from the iterative specification $G_{F}(D)$; i.e.,

Proposition 7 *Let $\tilde{B}$ be the behavior of the DEVF D and $\tilde{B}'$ be the behavior of $IOFO(G_{F}(D))$. Then,*

$$\tilde{B}' \supseteq \tilde{B}.$$

3.3 Free-Constructed DEVS

The Free-constructed DEVS [AZ93] is a discrete-event I/O system derived from *ideal* input/output data.

Based on the assumption that *all* input/output data and their corresponding initial states are available, a discrete-event IOFO specification can be constructed. By imposing two additional assumptions on the IOFO structure — time-invariance and causality — we can arrive at the desired IOFO, denoted as $IOFO_{ideal}$. Then we can derive, in an axiomatic manner, the Free-constructed DEVS, denoted as $DEVS_{free}$. This is a discrete event I/O system specification where precise transition functions, a time advance function, and an output function are defined for the input/output and state sets. That is, given $IOFO_{ideal} = \langle T, X, \Omega, Y, F \rangle$ we can construct $DEVS_{free}$ as:

$$DEVS_{free} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \qquad \text{where:}$$

1. The input and output sets are the same as those given in IOFO.
2. The sequential state set $S \subseteq F \times \Omega$ is defined by:

   $(f, \omega_{t>}) \in S \longleftrightarrow \omega_{t>}(t) \in X \quad \text{or} \quad f(\omega_{t>})(t) \in Y.$
3. The time advance function $ta : S \longrightarrow \Re^{+}_{0,\infty}$ is defined by:

   $ta(f, \omega_{t>}) = min\{e \mid e > 0, f(\omega_{t>} \circ {}_{e}\phi)(t + e) \in Y\}.$

   where ${}_{e}\phi$ is a segment, and $ta(f, \omega_{t>}) = \infty$ if no minimum element exits.
4. Total state: $Q = \{(s, e) \mid s \in S \text{ and } 0 \leq e \leq ta(s)\}$.
5. Internal transition function: $\delta_{int}(f, \omega_{t>}) = (f, \omega_{t>} \circ {}_{ta(f,\omega_{t>})}\phi)$.
6. External transition function $\delta_{ext}(f, \omega_{t>}, e, x) = (f, \omega_{t>} \circ {}_{e}X)$.

7. Output function: $\lambda(f, \omega_{t>}) = f(\omega_{t>} \circ \, {}_{ta(f,\omega_{t>})}\phi)(t + ta(f, \omega_{t>}))$.

The definition for segment ${}_e\phi$ (or any other segment similar in form such as ${}_eX$) is:

$$ {}_e\phi : (0, e] \longrightarrow X^{\phi} \qquad \text{such that} \qquad {}_e\phi(t) = \phi \qquad \text{for} \qquad t \in (0, e], \;\; e \in \Re. $$

Theorem 4 *Let $IOFO_{ideal} = \langle T, X, \Omega, Y, F \rangle$, and $DEVS_{free}$ be a free constructed DEVS from $IOFO_{ideal}$. Then, the input/output relations of $DEVS_{free}$ and $IOFO_{ideal}$ are equal [AZ93]; i.e.,*

$$ R_{IOFO_{ideal}} = R_{IOS(G_S(DEVS_{free}))}. $$

The above theorem asserts that $IOFO_{ideal}$ and $DEVS_{free}$ have the same input/output behavior. First, a free constructed DEVS (which is an I/O system) is derived from a causal time-invariant IOFO that contains all possible data. Then, a free iterative specification of $DEVS_{free}$, denoted as $IOS(G_S(DEVS_{free}))$, is derived by constructing $G_S(DEVS_{free})$ followed by $IOS(G_S(DEVS_{free}))$. Then, the input/output relations of $IOFO_{ideal}$ and $IOS(DEVS_{free})$ are shown to be equal.

Obviously, $DEVS_{free}$ is a deductive model. The assumption that we have all possible data allows us to construct the complete sequential state set, and consequently, the free iterative I/O system DEVS. Due to lack of an ideal data set, we need to operate at the IOFO level specification, thus allowing us to work with a finite data set. Unlike the ideal data set, a finite data set requires partitioning of trajectories and assignment of initial and final states in hope of predicting output histories that do not belong to the data set.

Chapter 4 AI and Non-Monotonic Reasoning

I have already discussed the need for making tentative choices when defining the free iterative IOFO specification due to lack of complete knowledge about the length, granularity, and possible state assignments to segments. Also, when there is no exact match for a given input segment, I proposed a set of assumptions to hypothesize about candidate segments.

System theory provided us with the basic means for representing input/output models (all variations of IOFO specifications). So it is fair to ask why I cannot use the same tools for reasoning. Unfortunately, the mathematics of system theory are too rigid to allow explicit forms of reasoning when knowledge cannot be assumed to be fixed and apriori.

Reasoning can be carried out by several schemes, each of which primarily follows one of two basic approaches: *probabilistic reasoning* [Pea88] and *logic-based reasoning* [GN87].

The former can be split into extentional and intentional. Intentional approaches to probabilistic reasoning, which are also referred to as model-based reasoning approaches, compute conclusions within the framework of probability theory [Lah79]. Extentional approaches (also referred to as production systems), however, use methods based on less stringent calculi such as the Dempster-Shafer certainty-factor calculus [Dem68, Sha79, Sho76], and fuzzy logic [Zad75, Zad81].

Logic-based approaches do not deal with the notion of certainty as it is defined for other approaches. They support reasoning based on the truth values TRUE, FALSE, and UNKNOWN assigned to entities to be reasoned with. They provide for a basic and simple type of reasoning, making tentative decisions that are subject to revision.

Several non-monotonic reasoning (NMR) theories have been formalized based on this form of reasoning. The features of other formalisms — usually absent in logic-based approaches — have, however, been incorporated into several of its variants [GN87, McC90, Nil86, Kir91].

One of the main concerns of this chapter is to bring to attention the issues involved in automating the processes of reasoning. It is paradoxical that, while NMR seems quite simple (at first sight), formalizing a general theory of it seems to be rather difficult. By briefly describing three early NMR formalisms, I will point out some of the NMR key characteristics and difficulties.

4.1 Logic and Knowledge Representation

Having decided on a logical approach to reasoning, I will briefly point out some key issues in using mathematical logic, which must be provided by any language formalizing logical approaches to reasoning.

For an intelligent entity to be able to reason, it must have an appropriate representation scheme for encoding its knowledge, as well as an inferencing mechanism to make decisions. Both of these issues (knowledge representation and reasoning procedures) have been treated extensively in the literature [End72, CL73, GN87, Ric83, Gin93]. I shall briefly describe each of these and their relationships to mathematical logic.

4.1.1 Propositional and First-Order Calculi

A language that can support NMR needs to be flexible. Knowledge can be represented using Propositional Logic, First-order (or Higher-order) Logic, and Modal Logic, among others. The implementation of logic-based truth maintenance systems is commonly based on a variation of First-Order Calculus, which is somewhat stronger than Propositional Calculus. I will present a short overview of propositional and first-order calculi that will become necessary in later discussion of some types of NMR.

Propositional Calculus:

The language of Propositional Calculus (PC), or Propositional Logic, is very simple both in terms of its syntax and semantics. However, the types of sentences that can be described in this language are quite limited in their expressive power.

Syntax specifies how sentences can be constructed according to a set of (syntactical) rules. Each sentence can be composed of logical symbols and sentence (or non-logical) symbols. Each sentence symbol represents a distinct object, such as A, that corresponds to an object in the universe of discourse (the set of objects about which the knowledge is being expressed). The logical symbols consist of (,) and the connective symbols $\neg$, $\vee$, $\wedge$, $\rightarrow$, and $\leftrightarrow$. Sentences constructed from these symbols according to grammatical rules are called *well-formed formulas* (or *wffs*). Grammatical rules specify the construction of (grammatically) valid sentences. Sentences are constructed recursively (cf. [End72] for details):

1. Every sentence symbol is a *wff*.
2. If α and β are *wffs*, then (α), $(\neg\alpha)$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $(\alpha \rightarrow \beta)$, and $(\alpha \leftrightarrow \beta)$ are also *wffs*.
3. No expression is a *wff* unless it is either of the form (1) or (2).

The semantics of propositional logic are straightforward. *Semantics* are concerned with truth assignment, i.e., the interpretation of sentences. Since this language is concerned with objects alone, non-logical symbols can be interpreted as being either true or false. In particular, a sentence is either satisfiable, valid, or invalid. A *satisfiable* sentence is true at least under one interpretation, while a *valid sentence* is true under every interpretation. A sentence that is not satisfiable is invalid.

Furthermore, the truth assignment for any finite set of *wffs* is finite. As a consequence, the language of propositional logic is decidable and thus tractable [GJ79, LP81].

First-order Propositional Calculus:

The language of First-Order Predicate Calculus (FOPC), or First-Order Predicate Logic, allows the construction of sentences that mention not only objects but also their properties in the world. It mirrors additional facets of natural language that make it much more powerful than the calculus of propositional logic for purposes of knowledge representation.

Besides the symbols of propositional logic, its syntax entails variable symbols and four additional parameters: the universal quantifier symbol $\forall$, constant symbols, function symbols, and predicate symbols. The universal quantifier symbol has the intended interpretation *for all.* Constant symbols name specific elements of a universe of discourse. A function symbol designates a function on members of the universe of discourse. Similarly, a predicate symbol designates a relation on members of the universe of discourse. For convenience, a *term* (i.e., a variable symbol, an object symbol, or a functional expression) is used to name an object in the universe of discourse.

Using the defined symbols and parameters, *wffs* for first-order logic can be constructed using the syntactical rules of first-order logic. In particular, there are three types of sentences: *atomic*, *logical*, and *quantified* sentences. The construction of *wffs* is defined recursively, as this was done in the case of the propositional logic.

An atomic sentence is formed from an n-ary predicate relation symbol ϕ and n terms $\tau_1, \ldots, \tau_n$ by combining them as $\phi(\tau_1, \ldots, \tau_n)$. The other two types of sentences are formed by using the logical symbols and the $\forall$ parameter. A logical sentence is constructed by combining atomic sentences with logical connectives according to the syntactical rules specified for propositional logic. A universal quantified sentence is constructed as $\forall_v(\phi)$, where ϕ is a simpler sentence and v is a variable[1].

Some additional terminology will be useful. There are other special types of sentences defined according to the type of variables they contain. A variable in a sentence is *free* if it is used in a term without an enclosing quantifier. A variable that occurs in a sentence and in an enclosing quantifier is called a *bound* variable. that has no *free* variable is called a *closed* sentence. A *ground* sentence contains neither free nor bound variables.

Unlike propositional logic, the semantics of first-order logic are not straightforward due to the presence of variables. In first-order logic, if an interpretation $\mathcal{I}$ satisfies a sentence ϕ for all variable assignments, then interpretation $\mathcal{I}$ is called a *model* of ϕ. Consequently, the notion of validity for a sentence no longer depends solely on the interpretations of a sentence symbol, but also on the variable assignments. A sentence is said to be *valid* iff it is satisfied by every interpretation and variable assignment. Similar to the definition of satisfiability of propositional logic, a sentence is called *satisfiable* if it has at least one model.

The notion of interpretation is quite important if a reasoning mechanism is to do reasoning, it has to rely on the syntax of the language alone. That is, it has to rely on a proof procedure to derive conclusions. A *proof procedure* derives all possible conclusions by using one or more inference rules, such as modus ponens, and a set of logical axioms. A *logical axiom* is satisfied by all interpretations because of its logical form.

However, when we are talking about something, we have in mind a particular

[1] Since $\forall \equiv \neg\exists$, there is no need to include $\exists$ in the language as an additional element. However, this can be done for convenience. Similarly, the smaller set of logical symbols $\neg$, and $\rightarrow$ would suffice to express all sentences in propositional logic, yet it is convenient to add the additional operators for the benefit of shorter sentences [End72].

Also, note that only variable symbols are allowed within the scope of the $\forall$ quantifier for a first-order language.

interpretation for symbols in our language. Therefore, it is necessary to show that a proof procedure (or our desired reasoning mechanism) infers facts that have the intended interpretation, i.e., it is necessary to show that the conclusions derived by a reasoning mechanism have the intended interpretation. We need to bridge the syntactical and the semantical notions.

Given a set of sentences Δ and a sentence ϕ, it is said that Δ *logically implies* (entails) ϕ, iff every interpretation of each sentence symbol that satisfies the sentences in Δ also satisfies ϕ. The logical implication is denoted as $\Delta \models \phi$.

Since, in general, we cannot specify a particular interpretation, it becomes necessary to derive conclusions (implicit facts) that are true in *all* interpretations. Therefore, a reasoning mechanism does not have to know which interpretation is intended as long as it adheres to this constraint.

An inference rule such as *Modus Ponens* can infer hidden (implicit) facts (sentence symbols) from what is already known. A proof procedure (which applies inference rules to the sentences in a database) is said to be *sound* iff any sentence that can be derived from a database of facts using it, is logically implied by that database. Also, a proof procedure is said to be *complete* iff any sentence logically implied by a database can be derived using that proof procedure.

More definitions are called for. When there exists a proof of a sentence ϕ from a database Δ and the logical axioms using modus ponens, ϕ is said to be *provable* from Δ. A sentence ϕ derived from Δ, denoted as $\Delta \vdash \phi$, is called a *theorem.* A *theory* is a set of sentences closed under logical implication. A theory, $\mathcal{T}$, is said to be *complete* iff for any sentence ϕ, either ϕ or $\neg\phi$ is a member of $\mathcal{T}$.

In the development and formalization of a reasoning formalism such as NMR, it is helpful to use the notion of logical implication to study and verify the correctness of the properties of a proof procedure. Model Theory provides the bridge to study the relationship between a syntactical object (a proof procedure) and its semantical counterpart (its interpretations) [CK90, Bri77].

For instance, a proof procedure that uses a set of logical axioms and modus ponens is shown to be semidecidable. This means that there are sentences that are entailed (logically implied) from a set of sentences, but cannot be inferred by this proof procedure. However, a proof procedure that is equivalent to a logical implication, denoted as $\Delta \models \phi \equiv \Delta \vdash \phi$, is said to be decidable. That is, any sentence that is logically implied by a set of sentences can be inferred by the proof procedure. These notions are necessary in our forthcoming discussions of NMR approaches.

4.1.2 Inference and Resolution Principle

As mentioned earlier, our intended reasoning mechanism needs to be a proof procedure of some kind. It is known that a proof procedure based on a set of axioms and modus ponens is semidecidable [GN87]. We can use the *resolution procedure* to ensure decidability for a special class of *wffs*. It uses the *resolution principle* as its rule of inference [Rob65, Rob79]. However, the resolution principle operates on *clauses*, expressible in a simplified variant of first-order logic, called *clausal form* logic. A first-order logic's closed sentence can be transformed easily into its clausal form using a procedure.

A clausal form has symbols, terms, atomic formulas, literals, and clauses. The difference between a clausal form and a *wff* is that a clausal form has literals and clauses instead of logical and quantified sentences [GN87, Ric89]. A *literal* is an atomic sentence or its negation. An atomic sentence is called a *positive* literal, whereas its negated form is called a *negative* literal. Having these, a *clause* is defined as a disjunction of a set of literals.

It should be noted that a set of clauses is equivalent to sentences of first-order logic in the sense that a sentence is satisfiable by a set of sentences iff the corresponding set of clauses is satisfiable. That is, nothing is lost in the transformation processes.

For so-called *Horn clauses* [GN87, Ric89], the resolution principle is both sound and complete. A Horn clause is a clause with at most one positive literal. A *Definite clause* is a more restricted clausal form. It contains at least one positive literal.

However, in general, a proof procedure that is designed to operate on sentences of first-order logic may not generate the same results if clausal forms are used instead.

The idea behind the resolution principle is simple: if we know that either P or Q is true and if we also know that P is false or R is true, then either Q or R is true. That is, given two clauses Φ : $\{\mathtt{P},\mathtt{Q}\}$ and Υ : $\{\neg\mathtt{P},\mathtt{R}\}$, then the clause $\{\mathtt{Q},\mathtt{R}\}$ is the resultant. Or in general, when no variable instantiation is necessary, the resolution principle can be represented as:

$$\begin{array}{ll} \Phi & \textit{with} \quad \phi \in \Phi \\ \Upsilon & \textit{with} \quad \neg\phi \in \Upsilon \\ \hline (\Phi - \{\phi\}) \cup (\Upsilon - \{\neg\phi\}) & \end{array}$$

A definition of the resolution principle that is applicable to any literal, with or without variables, can be given [GN87]. However, the above definition is sufficient for our purposes.

Having defined the resolution principle, a proof procedure (also referred to as resolution deduction) can be defined. A resolution deduction of a clause Φ from a database Δ is a sequence of clauses in which (a) Φ is an element of the sequence, and (b) each element is either a member of Δ or the result of applying the resolution principle to clauses earlier in the sequence[2].

[2] If the resolution principle is used unconstrained, the resolution deduction becomes inefficient.

4.2 Non-Monotonic Reasoning

We rarely realize the degree of sophistication involved in the reasoning processes we use to make our everyday decisions. Walking through a crowded street, for example, requires a tremendous amount of reasoning. To avoid colliding with other pedestrians as well as obstacles, one makes tentative instantaneous decisions to take a step a little faster, squeeze through a few people, bend a little to walk under a canopy. Students at the University of Arizona walking in the Student Union hallway often decide at a moment's notice to change their walking paths; as they are changing their paths, they might have to resort to alternate paths to avoid unexpected encounters. What is important is that a student is making a tentative decision, and is prepared to revise it as soon as there is some reason to do so[3]. This type of decision making is necessary for us to maneuver our way through a world undergoing continual change[4].

Since its early days, one of the grand goals of the field of AI has been to automate this type of decision making [McC58, McC90].

Reasoning can be either monotonic or non-monotonic. As the name indicates, *monotonic reasoning* [CL73, GN87, Gin93] only allows making decisions that cannot be revoked later. *Non-monotonic reasoning* [McC80, Rei80, MD80, Bob80, Bes89, Bre91, MT93], which manifests itself in many types of problems, allows making tentative decisions, subject to future revocation. That is, it allows us to back off from an earlier decision (thought) once there is evidence justifying it. What should be noticed is that the evidence that supports this altered decision

For this reason, various strategies, such as unit resolution and linear resolution, are used to improve the efficiency of the resolution deduction [GN87]. Such proof procedures are called *brief deductions* [McC62, MW91].

[3] There is a nice example in [Pea88]. It compares the tension between intentional and extentional systems faced with the problem of crossing a minefield on a wild horse. If, on the one hand, the horse is restrained too much, it won't go anywhere; on the other hand if you let it loose, the fast ride may end up with a disaster. This argument can be analogously used for a pedestrian strolling in a crowded street.

[4] [Bur90] argues that, in fact, most of human reasoning is non-deductive.

was not available at the time the original decision was made.

Just as NMR plays a vital role in our daily lives, any proposed intelligent machine needs to support NMR as well [MH69, Nil91, GN87, Bre91, MRS88, McC90]. The main reasons for requiring this form of reasoning can be traced back to two fundamental problems — the *qualification problem* and the *frame problem* [MH69].

The qualification problem refers to trading the amount of knowledge required for making inferences for the accuracy of these inferences. In a real-time environment, decisions must be reached within a finite, and often quite limited, amount of time. It may not even be feasible to review the entire body of currently available information when making a decision. Thus, decision making under uncertainty may be a necessity even in situations where complete information would theoretically be available. It is this difficulty that makes NMR necessary, since we cannot possibly expect humans (or machines for that matter) to be aware of all the preconditions that might trigger, enable, or inhibit a decision.

The frame problem is concerned with how to avoid specifying axioms about which actions do not interfere with other actions. A famous problem from the Block World makes this clear. Suppose, in state s_1, block b is red and is on the table. Also in state s_1, block a is on the (same) table with block c placed on top of it. Now, if block c is moved and placed on the table, resulting in state s_2, then it is fairly easy for us to conclude that block b is still red. To date, no mathematical formalism can make this conclusion without being provided with appropriate axioms. The difficulty lies in the fact there are numerous axioms that have to be specified before a machine can say block b is still red.

The frame problem is a more general form of the qualification problem; consequently, it seems to be much harder to solve. In this work, I am concerned with the qualification problem only.

The kinds of tasks (e.g., language understanding and reasoning) that caught the attention of the early AI community suffer from these two problems in various disguises. However, to most people, language understanding and reasoning seem rather trivial, since they are quite successful in carrying them out. To the surprise of many, these deceivingly simple-looking tasks have proven exceedingly difficult to formalize [GN87, Gin93, Ric83, Sho88].

4.3 Limitations of Monotonic Reasoning

What underlies *monotonic reasoning* is the use of one or more *sound* rules of inference, each consisting of (1) a set of sentence patterns (conditions), and (2) another set of sentence patterns (conclusions). Typical monotonic rules of inference are the resolution principle and modus ponens. Classical logic rests on the idea that its inference rules are sound. While a sound inference rule ensures deduction of (implicit) conclusions that are consistent with what is already known, it can produce no *new* knowledge. Furthermore, adding new sentences to the database *cannot* invalidate any of the previously derived conclusions.

This strict form of monotonic inference rules does not allow the expression of tentative statements. Another difficulty with monotonic inferencing is the qualification problem — an inference rule is applicable only if its premises (conditions) are known.

To overcome these difficulties, one can use *non-monotonic* inference rules [GN87, Bes89, Bre91, MT93]. With such inference rules, an earlier conclusion may have to be retracted if another sentence is added to the database. These inference rules are generally called *non-monotonic.* That is, for a given database Δ, we allow non-monotonic inference rules to be used to infer *new* conclusions that do not logically follow from Δ when using sound inference rules only. Basically, NMR approaches attempt to formulate (1) non-monotonic inference rules and/or (2) a formalism that can support well defined, yet revocable, inferences.

4.4 Non-Monotonic Reasoning Logics

Despite the widespread usage of NMR in our own decision making, no serious efforts had been made until recently to incorporate such mechanisms into algorithms. Although the need for understanding and formalizing the NMR style of reasoning was realized quite a long time ago [McC58, MH69], its first theoretical accounts weren't reported until late 1970's [Rei78, Cla78, Doy79, McC79].

Different taxonomies of NMR logic have appeared in the literature [Bre91, GBS93, Bes89, Gin87, Sho88, MT93]. However, I prefer to present a somewhat different categorization. I take the point of view that NMR can be approached from two perspectives: the experimentalist approach and the formalist approach. Yet, there is a need to proceed from the theory developed by formalists to something that is actually computable. Likewise, it is necessary to put on a firm ground the implementation of a non-monotonic reasoner. Much work has been done to go from one direction to the other, and vice versa [MRS88, GBS93].

I begin by classifying the NMR approaches from the formalists' point of view. I consider some mathematical formalisms that have been developed without taking into account their computational counterparts. My motive is to bring to attention the issues involved in building a non-monotonic reasoner. In this respect, I choose to distinguish one class from another by identifying the type of logic, such as model-preference logic and fixed-point logic, underlying the decision making of the reasoner.

The main idea behind *model-preference logic* is that one interpretation is preferred over another. For instance, we may want to choose an interpretation that minimizes the number of positive facts in a given theory. Close-World Assumption (CWA), Predicate Logic, and Circumscription are three formalisms belonging to this class [GN87, Rei78, Cla78, Bob80, McC80].

Fixed-point logic supports NMR by choosing from a set of available default assumptions, the maximum number of assumptions that are consistent among each other and that can be added to the theory. Since this class of logic can best be formalized using the idea of a fixed-point, it is known as fixed-point logic. The main approaches belonging to this class are Default Logic [Rei80], Modal Non-monotonic Logic [MD80, McD82], and Autoepistemic Logic [Moo85].

One important feature distinguishing these two approaches from one another is that model-preference logics do not devise new logics. Instead, new non-monotonic inference rules are introduced for making tentative inferences. Fixed-point logic, however, devises new logics such as Default Logic.

In this work I choose the model-preference approach. This is because (1) classical logic offers a richer base of concepts and mechanisms that are more applicable to model-preference logic than to fixed-point logic, (2) the types of problems that I am interested in are amenable to model-preference logics at least as easily as to fixed-point logics, and (3) there still exists a better pool of computational models[5] and tools for model-preference logic, both from the point of view of existing classical theorem provers and the ease with which they are likely to be enhanced in the future.

Next, I discuss a couple of model-preference approaches that can meet my present needs, and seem powerful enough to satisfy foreseeable additions. Computational models have already been devised for some variations of these model-preference approaches [FdK93, Gin89, MRS88]. In particular, I shall briefly describe Closed-World Assumption and Predicate Completion, and shall add a few words on Circumscription. These formalisms are all expressible within classical logic, and thus no new logic (e.g., Default Logic) is necessary.

One apparent difficulty with NMR is that its intuitive definition is too vague! Consequently, there are many misconceptions about what NMR really is. The

[5] For the last several years, an entire research community has devoted much effort to Logic Programming as an alternative means for computational models of non-monotonic logics.

reason may be that we are all too familiar with the notion of belief revision. This familiarity, unfortunately, lures people into using the term too loosely. It is used indiscriminately to refer to backtracking in trees and graphs, updating databases, diagnosis, qualitative modeling, synthesis problems such as deign and planning, reasoning about actions, and common-sense reasoning, among others.

The most serious misconception is that non-monotonic reasoning is equivalent to backtracking. NMR is much more than backtracking! In traversing a tree, for example, a backtracking mechanism provides the ability to backtrack a path. However, the path itself usually remains static. As NMR backtracks a chain of inferences, it is providing a foundation for creating dynamically a chain of inferences and, consequently, withdrawing part or all of that chain of inferences as necessary while new knowledge is being added. That is, although the behavior of NMR resembles that of backtracking — and indeed, NMR does backtracking — it is a much more powerful notion. The following should elucidate some basic features of NMR. More importantly, various NMR formalisms provide some means for the extension of theories.

The order in which I discuss the following three formalisms has the merit that, as I proceed from one to the next, each formalism becomes more powerful than the previous one in the sense that the sophistication of non-monotonic reasoning increases by allowing the use of more general logic sentences.

4.4.1 Closed-World Assumption

The qualification problem amounts to having to specify too many things. For instance, a database's only entries may be pairs of neighboring countries. It does not have any entries for non-neighboring countries. Intuitively, we could conclude that, if two countries are not listed as neighbors, then they are not — although this may not be true if we find out later that this entry had simply been left out of the database. This kind of conclusion cannot be made with classical logic because the theory for this database is not complete. In a complete theory, either a ground sentence or its negation must be in the theory. As mentioned earlier, the qualification problem is a battle between specifying too many and too few facts.

In databases with many entries it is very useful to only include either the set of positive facts or that of negative facts, whichever is smaller. Then, if something is not in the database, we conclude its negation with the provision that we may have to withdraw this conclusion later and augment the theory as necessary. The Close-World Assumption (CWA) approach simply completes the theory by adding tentative conclusions to Δ [Rei78, She84, GN87].

CWA can be clearly described in terms of a database, Δ, containing some axioms called *facts*, a theory, $\mathcal{T}[\Delta]$, which is the closure of Δ under logical implication, and another database, Δ_{asm}, containing some axioms called *beliefs*. We know that the *wff* $\phi \in \mathcal{T}[\Delta]$ iff $\Delta \models \phi$. Then, each element of the belief dataset, Δ_{asm}, can be derived based on CWA provided that $\neg \mathrm{a} \in \Delta_{asm}$ iff the ground atom $\mathrm{a} \notin \mathcal{T}[\Delta]$, that is, $\phi \in \mathrm{CWA}[\Delta]$ iff $\{\Delta \cup \Delta_{asm}\} \models \phi$. Hence the augmented theory CWA[Δ] is the closure of all facts and beliefs.

Unfortunately, in general, CWA does not produce consistent results for all types of theories. However, CWA augmentation is consistent for the important class of Horn theories. We also don't want to assume that any ground sentence that is not provable from Δ is false. In particular, we might want to weaken this assumption by considering CWA for a subset of predicates only. This limits the number of negative ground sentences that can be assumed false. However, while CWA augmentation is consistent for a single predicate, it may be inconsistent for a set of predicates. Two other types of assumptions — the Domain-Closure Assumption (DCA) and the Unique-Name Assumption (UNA) — can also be used. Whereas CWA does not limit the constants of the language that occur in Δ, DCA does. By assuming

DCA, the objects in the domain are limited to those that can be named using the object and function constants occurring in the language. The UNA assumes two ground terms to be distinct, unless they can be proven equal.

4.4.2 Negation as Failure

It would be better if we could express what CWA is in terms of some sentences that can be expressed in one of the logical languages. The nice thing is that often a single logical sentence can express an assumption indicating that the only objects that satisfy a predicate are those that *must* do so given a set of beliefs. An example makes this clear. Let us assume that our database Δ contains a simple formula p(A). Then the following expression is equivalent to p(A);

$$\Delta \equiv \forall \mathtt{x}\ \ \mathtt{x} = \mathtt{A} \Rightarrow \mathtt{p(x)}$$

However, if we want to say that there is no other object satisfying p, we should rewrite the above expression as:

$$\forall \mathtt{x}\ \ \mathtt{x} = \mathtt{A} \Leftrightarrow \mathtt{p(x)}$$

The last expression (called the *completion of* p *in* Δ) is obtained by adding the expression (called *completion formula* for p)

$$\forall \mathtt{x}\ \ \mathtt{p(x)} \Rightarrow \mathtt{x} = \mathtt{A}$$

to the first expression [Cla78, She84, GN87]. Therefore, we say that the completion of p in Δ is:

$$COMP[\Delta;\mathtt{p}] \equiv (\forall \mathtt{x}\ \ \mathtt{p(x)} \Rightarrow \mathtt{x} = \mathtt{A}) \wedge \Delta.$$

Again similar to CWA, this approach is applicable for certain classes of formulas only. For example, predicate completion is shown to work for a set of clauses referred to as *solitary* clauses. That is, if we have a consistent Δ of solitary clauses in predicate p, then the completion of p in Δ is consistent [GN87]. For a set of solitary clauses in predicate p, each clause with a positive occurrence of p has at most one occurrence of p. Comparing Horn clauses and solitary clauses indicates that, whereas solitary clauses are necessarily Horn clauses as well, the reverse does not have to be true.

Similar to CWA, predicate completion for multiple predicates can be performed in parallel with proper handling of solitary clauses to avoid inconsistent completion of Δ with respect to a set of clauses. I do not describe this and other variants of Predicate Completion since my purpose is limited to pointing out the issues involved in formalizing the logic of Negation as Failure. A nice treatment of Predicate Completion is presented in [GN87].

4.4.3 Circumscription

The logic of CWA and Negation as Failure are attempts at augmenting their respective belief sets. Whereas CWA augments its belief set by including the negation of ground positive literals that cannot be proven, the logic of Negation as Failure deals with predicates instead. That is, Predicate Completion augments an existing set of solitary formulas with some new formulas, which collectively state that the only objects that satisfy the predicates are the ones that must do so. What is common between these strategies is that the idea of augmentation is based on a kind of minimization. That is, in one strategy, literals are minimized; in the other, predicates are minimized.

Another formalism called *Circumscription* [McC80, McC86, Lif85, Lif86, Lif87, Lif89] minimizes the so-called abnormal objects of a theory. In its basic form, it is a minimal conjecture rule, instead of an inference rule, expressible within classical logic. Like the previous two strategies, it is based on the minimization idea. Hence its goal is to minimize the number of objects that can be shown to be abnormal.

The general formulation of circumscription is a second-order formula that can be added to a theory with some predicates to be minimized with respect to some objects.

The circumscription formalism, however, is much broader and more powerful than the previous two formalisms. For instance, the canonical example is about a database containing facts about objects which may or not fly. Circumscription is used to express the assumption for only those objects that can fly. That is, there exist positive expressions about those birds that can fly and no statements about elephants that cannot fly. Therefore, it is possible to add new expressions about an elephant (a bird name may be Elephant) without affecting the previous expressions. The power of Circumscription is due to the possibility of adding expressions with exception handling, again without affecting the previous expressions. These expressions have much broader implications compared to expressions stating some facts or beliefs. I don't go into the mathematical discussion description of Circumscription and its ramifications, as I am interested in a much less sophisticated type of NMR that can be handled with CWA. A detailed exposition of Circumscription is provided in [GN87].

The purpose of discussing CWA as well as touching on Predicate Completion and Circumscription is to add precision to the framework of reasoning that I am committing myself to. Furthermore, it is useful for putting into proper perspective the material developed in the remainder of this chapter and in subsequent chapters.

4.5 Truth Maintenance Systems

In the previous section, I discussed some NMR formalisms that were based on the language of logic for representing knowledge and using some theorem proving techniques for manipulating belief. Here, I focus on an approach referred to as *Truth Maintenance Systems*[6] (TMS). It was developed and advocated primarily by the experimentalists [SS77, Lon78, Doy79, dK86b, McA90, MR91], but recently, some formalists have also become interested in TMS systems [RdK87, McD91, NR90, MT93].

A problem solver has to generate *acceptable* responses given some inputs and enough knowledge about the tasks at hand. Why do I use the term acceptable in the previous sentence? The reason is that the kind of problems for which this type of problem solvers is useful implicitly takes into account the fact that problem solvers are inevitably faced with situations where they have to make tentative decisions, knowing there is not enough information to warrant a decisive one. Thus, I am primarily concerned with problem solvers that do have *sufficient* partial knowledge. The qualifier "sufficient" is used to make clear that at least a minimum amount of knowledge has to be made available to a problem solver if it is to generate any acceptable responses.

A problem solver has to keep track of all the decisions it makes based on incomplete knowledge. This explains why the problem solver is partitioned, as shown in Figure 4.1.

The need for Truth Maintenance Systems is the same as before: there are situations where some tentative decisions have to be made based on unsupported knowledge (also referred to as assumptions). This implies that decisions may have

[6] In the literature sometimes the better suited names of Belief Maintenance Systems or Reason Maintenance Systems are used instead.

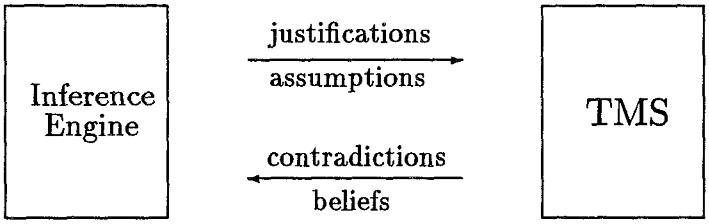


Figure 4.1, Caricature of a problem solver.

to be made with the authority to withdraw any incorrectly hypothesized decisions once new facts that contradict the assumptions become available. Since revocation of these temporary decisions should be efficient, it is quite useful (and sometimes necessary) to devise not a simple backtracking, but a rather more powerful scheme such as *dependency directed backtracking* [SS77, Lon78, Ric83, Gin93, FdK93]. The roots of the evolution of TMSs are based on dependency directed backtracking principles.

It is useful to define a problem solver — or, more appropriately, a reasoning system — as comprised of an inference engine and a TMS [dK86b, FdK93] (cf. Figure 4.1 [FdK93]).

The task of keeping and manipulating records of inferences and their proofs is assigned to a Truth Maintenance System. The inference engine provides inferences and their proofs to the TMS[7]. This partitioning of a problem solver provides better control as well as more efficient computations for most problems. Nevertheless, the dividing line of responsibility between an inference engine and its corresponding TMS is not crisp. For instance, some inferences to be carried out by an inference engine can be delegated to the TMS. In fact, several types of TMSs (cf. Figure 4.2)

[7] The name TMS is sometimes used to refer to a TMS as well as to the problem solver itself.

are possible, partly because the tasks of a problem solver are divided between the inference engine and the TMS.

One thing to note about this approach is that a TMS can be largely independent with respect to the type of inference engine. For instance, the inference engine could be a rule-based system or a theorem prover.

I shall now discuss, in a general setting, how the problem solver carries out its task. The inference engine and the TMS are exchanging information using a well-defined protocol. That is, the inference engine's important inferences are sent to the TMS as *justifications* (I shall shortly describe what a justification is). The TMS, having recorded these justifications, aids the inference engine by:

1. discovering inconsistencies between justifications;
2. finding out from justifications what set of assumptions leads to contradictions;
3. stashing information about failed inferences as well as successful inferences as justifications to avoid redundant work;
4. facilitating default reasoning by using TMS justifications to include explicit default assumptions; and
5. generating explanations by tracing through justifications.

In order to understand how the inference engine and the TMS interact, I need to define some concepts and terminologies. A problem solver's (or more precisely, an inference engine's) *datum* refers to its assertions, facts, inference rules, and procedures. Each datum has an assigned counterpart in the TMS called a *TMS node*. Both the TMS and the inference engine need to communicate with each other via TMS nodes. Therefore, each inference engine's datum points to its associated TMS node, and each TMS node points to its counterpart datum. The responsibility of establishing this connection is assigned to the inference engine. Although TMS nodes are communicated between the inference engine and the TMS, each TMS node is interpreted differently by the inference engine and the TMS. TMS nodes are used as data by the inference engine to make inferences. The same TMS nodes with assigned truth values are used by the TMS to maintain a consistent dataset. TMS does not use these nodes as data.

A node of a TMS can be either a *premise*, a *contradiction*, or an *assumption*. While a premise and a contradiction hold indefinitely, an assumption may hold temporarily, making it subject to revision as necessary. Each node, furthermore, can have a label to indicate the current belief in it. For example, in Logic-based TMSs, a node's label can be either TRUE, FALSE, or UNKNOWN.

A justification is essentially a deduction (or inference), which is a type of constraint on conditions on some TMS nodes. Each justification consists of three parts:

- Antecedents are the nodes of the data used as antecedents to an inference rule.
- A Consequence is a node corresponding to the data that is inferred using an inference rule and known antecedents.
- The Informant contains explanatory information associated with an inference rule.

The syntax of justification is:

$$(\langle \textit{Consequence} \rangle \ \langle \textit{Informant} \rangle \ \cdot \ \langle \textit{Antecedents} \rangle)$$

Now, we can view a TMS as a *dependency network structure* consisting of TMS nodes connected via justifications. Also, we can identify different families of TMSs (cf. Figure 4.2 [FdK93]) using the type of constraints the inference engine is allowed to express among nodes, as well as the kind of queries the TMS is expected to answer efficiently.

Three basic types of constraints are: Justifications (i.e., Definite or Horn clauses as defined in Section 4.1.2), Non-Monotonic Justifications, and Arbitrary Justifications. The node type could be either simple or complex. For example, where

	Simple Label	*Complex Label*
Horn/Definite Constraints	JTMS	ATMS
NM Clause Constraints	NMJTMS	—
Arbitrary Clause Constraints	LTMS	CMS

Figure 4.2, Families of TMSs

a simple node's label can be TRUE, a complex node's label may consist of the set of assumptions under which the node is TRUE. Therefore, as shown in Figure 4.2, a TMS can be a Justification-based TMS (JTMS), a Logic-based TMS (LTMS), a Non-Monotonic JTMS (NMJTMS), an Assumption-based TMS (ATMS), or a Clause Management System (CMS) [FdK93][8]. Our own problem solver uses a Logic-based Truth Maintenance System. Nevertheless, before discussing LTMSs in more detail, I wish to point out a few important aspects of JTMSs. A comprehensive exposition of JTMSs, LTMSs, ATMSs as well as their implementations is provided in [FdK93].

A JTMS only accepts justifications that can be expressed as definite clauses (i.e., $\alpha_1 \wedge \ldots \wedge \alpha_n \Rightarrow \beta$ where α_i and β are TMS nodes). A premise node is a justification with no antecedents. A contradiction node is specifically designated by the inference engine. The belief in a contradiction node does not interfere with the operations of the JTMS. They are there to be sent to the inference engine once they become believed; that is, contradictions have to be resolved by the inference

[8] In principle, every type of TMS is powerful enough to emulate the other kinds [McA90, McD91].

engine, which has to ensure that they are not believed.

A node can also be an assumption. Again, the inference engine has to designate it as such explicitly. An assumption is *enabled* if the inference engine chooses to believe it. Otherwise, an assumption is treated as any other node; that is, it can be believed if its justification is believed, or retracted if its justification is not believed. In a JTMS, a node is either "in" or "out," where *in* indicates that it is believed, and *out* indicates it is not.

Now I can describe briefly how the JTMS operates. We can formulate a JTMS within propositional calculus and hence have precise ideas about what the JTMS is actually computing. Every node is a propositional symbol $s \in \mathcal{S}$. The set $\mathcal{A} \subset \mathcal{S}$ denotes the set of assumptions. Then every justification can be viewed as a simple propositional definite clause, the set of which is denoted as $\mathcal{J}$. Within the setting of propositional calculus, a node α is *in* when it follows from $\mathcal{A} \cup \mathcal{J}$ under the rules of propositional calculus. Otherwise, the JTMS indicates that node α is *out*. The main advantage of the JTMS approach is that, since all justifications are definite clauses, a simple forward propagation algorithm can be used for implementing JTMSs. However, the price for this efficient implementation is that a JTMS is quite limited in its expressing power. For example, it cannot represent $\neg\alpha$ since it is not a definite clause without resorting to various encoding tricks.

Nevertheless, a JTMS can be used for providing well-founded support for those nodes that are believed as well as storing justification for efficiency. Also, it can support default reasoning with help from contradiction nodes, even though it lacks precise semantics.

4.5.1 Logic-based Truth Maintenance Systems

The additional expressive power of an LTMS can be quite beneficial. A Logic-based Truth Maintenance System permits expressing arbitrary propositional clauses (e.g., a negative node can be expressed just as easily as a non-negative node). It is possible to tinker with a JTMS to allow it to express any clause as well. Various encoding tricks can be used to overcome the JTMS's lack of expressive power in its natural setting. That is, it is possible to obtain much of the logical power of an LTMS by using a JTMS with the aid of appropriate encoding procedures. However, the end result is prone to more errors in terms of formulating the clauses. It also produces a dependency network that can require too many computational resources. Hence, having a TMS that can express arbitrary propositional clauses not only eliminates error-prone encoding schemes, but also provides better facilities for carrying out the TMS's tasks, such as assisting the backtracking process by eliminating surplus computations that would otherwise be necessary [FdK93].

Much of the top-level functionality of an LTMS is the same as that of a JTMS. The differences between them, aside from their expressive power, mainly lie in the algorithms that are used in carrying out the tasks of the problem solver.

A TMS can distinguish between several node properties. In a JTMS, a node can be either a premise, a contradiction, or an assumption. In an LTMS, there are only two node properties:

- A node is a premise if it is the only node in some inference engine supplied clause (the LTMS does not require an explicit premise property).
- A node is an assumption if the inference engine tells the LTMS that it chooses to label it as either TRUE or FALSE.

Contradiction nodes are meaningless in an LTMS. This is due to the way in which clauses are represented in clausal propositional calculus. For instance, in a JTMS, we may have a justification as:

$$\alpha \wedge \beta \;\Rightarrow\; \bot$$

where symbol $\bot$ indicates a contradiction. The above justification can be equivalently represented in LTMS as:

$$\neg\alpha \vee \neg\beta \vee \perp \;\equiv\; \neg\alpha \vee \neg\beta.$$

The last clause eliminates the need for including contradiction nodes. Contradiction literals in clauses cannot contribute in satisfying clauses.

Hence, a node in an LTMS is either a premise or an assumption. We can denote the set of clauses as $\mathcal{C}$ and the set of assumptions as $\mathcal{A}$. An assumption is a node the belief of which can be altered by an explicit inference engine operation. As in a JTMS, each node in an LTMS has a label; it can be either *known* or *unknown* since it is based on propositional calculus. If a node is known, it can either be TRUE or FALSE. Specifically, a node is labeled TRUE if it is derivable from $\mathcal{C} \cup \mathcal{A}$. A node is labeled FALSE if its negation is derivable from $\mathcal{C} \cup \mathcal{A}$. The label of an unknown node is UNKNOWN, in case it is neither TRUE nor FALSE, i.e., in case no arbitrary assumption was made about that node.

Note that in an LTMS, if a node has a label, there is no need to include a separate node representing its negation, as would be the case in a JTMS. The LTMS simply assigns the label FALSE or TRUE to the negation of a node's label that is TRUE or FALSE. Of course, if a node's label is UNKNOWN, then the label of its negation is also UNKNOWN. The ability of an LTMS to represent negative literals in terms of positive literals results in the creation of fewer nodes. Consequently, an LTMS's efficiency improves since a smaller dependency network is needed for fewer nodes.

Logical Specification of LTMS:

The dependency network for an LTMS is comprised of a set of nodes (or a set of literals) denoted as $\mathcal{S}$. A subset of these nodes, denoted as $\mathcal{A} \subset \mathcal{S}$, are assumption literals. Then, according to clausal logic, a clause is a disjunction of literals with no repeated literal. Having the literals, the inference engine can then supply a set of clauses $\mathcal{C}$ defined over $\mathcal{S}$. Again, we can assume that $\mathcal{C}$ grows monotonically, whereas $\mathcal{A}$ does not.

The three basic tasks of the LTMS are:

1. Provide a label for each node.
2. Detect contradictions.
3. Provide explanations for the label of each node.

To provide a label for s inquired by the inference engine, the LTMS can respond with one of the three possible labels TRUE, FALSE, or UNKNOWN. If there exists a $\mathcal{E} \subset \mathcal{C} \cup \mathcal{A}$, such that s follows from $\mathcal{E}$ propositionally, then the LTMS assigns the label TRUE to s. It is also possible that $\neg s$ is provable from $\mathcal{E}$. In this case, the label FALSE is assigned to s by the LTMS. If neither $\mathcal{E} \vdash s$ nor $\mathcal{E} \vdash \neg s$, then the LTMS assigns the label UNKNOWN to s.

The second task implies that somehow $\mathcal{C} \cup \mathcal{A}$ is unsatisfiable, else there would not be a contradiction to be concerned with. The only explanation for $\mathcal{C} \cup \mathcal{A}$ becoming unsatisfiable, is that the LTMS has arbitrarily assigned the labels TRUE or FALSE to one or more of the unknown nodes. Some of these arbitrary assumptions will now have to be revised in order to make $\mathcal{C} \cup \mathcal{A}$ again satisfiable. Luckily, the LTMS keeps explanations for these labels that provides the inference engine with the information necessary for resolving the contradiction.

The third responsibility of the LTMS is to provide explanations for nodes, even when $\mathcal{C} \cup \mathcal{A}$ is satisfiable. Typically, these explanations will be pointers to the assumptions that had to be made in order to reach a particular TRUE or FALSE value for a node. This feature is one of the primary benefits of equipping a problem solver with a Truth Maintenance System.

The above logical behavior of the LTMS can be implemented using a variety of algorithms. A particularly efficient algorithm is the so-called *Boolean Constraint Propagation* (BCP) algorithm. The BCP algorithm used in this work is restricted to clauses. However, as I mentioned previously, any propositional formula can be converted into a logically equivalent set of clauses.

The basic specification of a BCP [FdK93] is sound with respect to propositional calculus both in terms of labeling nodes as either TRUE or FALSE, as well as detecting contradictions only when they are logically provable. Unfortunately, the BCP's basic specification is not strong enough to ensure logical completeness[9]. For our purposes, it suffices to say that, for positive literals and Horn Clauses, the basic specification of the BCP can be extended to make it logically complete [FdK93].

[9] The logical completeness of a BCP can be viewed in terms of *literal completeness* and *refutation completeness*. The former refers to the ability of the BCP to label a node as either TRUE or FALSE when it is logically entailed from PC. The latter says that, whenever a contradiction is logically entailed from PC, the BCP can detect it.

4.6 Applications of NMR

The most challenging application of NMR seems to be in common-sense reasoning relying on common-sense knowledge. John McCarthy discusses his views on what *Common Sense* is. He refers to the term *Common-Sense Capabilities* and divides it into *common-sense knowledge* and *common-sense reasoning* [McC84]. The former refers to facts and the latter to some forms of inference.

The use of the term common-sense reasoning is not limited to deductive, inductive, or abductive modeling paradigms; it can be identified with *any* reasoning scheme that humans are capable of. We often say it makes *sense* to say something or do something, e.g., it makes sense for a ball thrown into the air to fall down after some time. It was this general usage of the term common-sense reasoning that first caught the attention of some early AI researchers [McC58, MH69, McC90]. One may argue that, in fact, common-sense reasoning applications encompass all others — applications based on probability theory and its variations, and fuzzy logic, for example. My motive is not to argue about such claims or delve into epistemological issues about what is and what is not common-sense reasoning. Instead, I would like to point out that deductive qualitative modeling is considered to rely extensively on common-sense reasoning [HM85, FdK93][10].

Default reasoning, inheritance, and reasoning about action are considered different variations of non-monotonic reasoning[11]. In applications that exhibit non-monotonic characteristics, the primary reason for using NMR can eventually be traced back to the need for dealing with the qualification problem. What is more interesting is that, although NMR is itself inductive in nature, to y knowledge NMR has never been used so far to deal with inductive forms of reasoning. All

[10] It is interesting to note that the term "qualitative modeling" is commonly used in the literature as equivalent with "deductive qualitative modeling," i.e., the attribute *deductive* is silently implied.

[11] [Lif88] has compiled a set of benchmark problems to aid researchers in the study of formalisms for non-monotonic reasoning.

reported usages of NMR relate to either deductive or abductive modes of reasoning.

For instance, NMR has been used in deductive qualitative modeling [HM85, FdK93, Gin93]. The inference engine for deductive qualitative modeling is built using deductive inference rules that essentially describe how implicit knowledge can be derived based on the axioms specifying how the system is supposed to work. These axioms, for instance, may describe how a motor or a thermostat is expected to operate under some well-defined conditions. Then, NMR is used to allow for properly handling situations that had not previously been expected, and for which consequently no provisions were explicitly specified. The role of NMR is to augment the existing theory, i.e., the set of axioms describing the behavior of the system, with additional knowledge such that the augmented theory is able to handle some unspecified situations as well. The inferencing mechanism of NMR is non-monotonic, and thus it is possible to withdraw decisions that are refutable given new axioms. Clearly both deductive and inductive inference rules are involved in deductive qualitative modeling.

In contrast, our own contribution to NMR relates to inductive reasoning modes. In inductive qualitative modeling, the situation is quite different. There are no theorems available that would describe the underlying behavior of the system. There exists no theory that can be subjected to deductive rules of inference to predict the system's behavior. Instead, the theory contains axioms expressing a system's observed behavior. That is, whereas in deductive qualitative modeling a set of theorems describes how the system is expected to operate, in inductive modeling there exist no such theorems. Indeed in inductive modeling, we are constraint with some finite observed dataset. We use some inferencing mechanism to make decisions. These decisions are purely based on data rather than on theorems expressing how the system is supposed to operate. Hence, given a finite set of data, applicable inference rules have to be non-monotonic[12].

Therefore, the use of NMR in inductive qualitative modeling is quite different from its use in deductive qualitative modeling or any other application that is formulated based on either deductive or abductive paradigms. In inductive modeling as I have defined it, we need to rely either on inductive (non-sound) or non-monotonic inference rules. Although there exists an extensive body of literature on inductive modeling and machine learning, to our knowledge, none of the previous research efforts in inductive modeling made use of non-monotonic reasoning.

[12] Note that inductive inference and non-monotonic inference are distinct.

Chapter 5 Discrete-event Inductive Modeling

A modeler has to inevitably consider a system to be modeled from a particular point of view [Min65, Zei76, Cel91, Wym93]. The basic idea is to view a system as delineated from its surroundings that externally interact with the enclosed system. The separation between the system itself and the part that lies outside of it provides a particularly powerful means to deal with the difficulties of modeling. Consequently, a modeler must decide what types of questions the model is expected to answer. Several other important choices, such as the levels of complexity and granularity of data (e.g., input and output), have to be made as well. A powerful concept called *abstraction* is particularly useful to effectively deal with such choices [Zei76, Zei84, Kli85, Wel86, CK90, GW92, Wym93].

The basis for modeling, and in particular inductive modeling, is abstraction. Reasoning is intrinsically intertwined with abstraction. The ability to use abstraction and reasoning simultaneously is very useful and powerful in modeling.

Abstraction allows one to deal with the *lack* of knowledge. As I mentioned in the previous chapter, the frame and qualification problems are manifestations of insufficient knowledge, hence indicating the eligibility of abstraction as a potent candidate to deal with these problems. In particular, my tenet is that the underlying ideas of NMR are based on applying some kind of abstraction to overcome the lack of knowledge.

For an abstraction to be useful, it must be the "right" one. Of course, the difficulty is determining what "right" means, i.e., what makes an abstraction effective? What does it ignore? What does it emphasize? Three basic measures for determining a "right" abstraction are validity, applicability, and computability [Zei84]. Obviously, much effort (reasoning) is needed before any of these questions can be answered effectively.

5.1 Abstraction in Modeling

Theories of abstraction, as defined by [Zei76, Wym93], provide the means for comparing behaviors of two systems, one being an abstraction of the other. One may be called a *big* system and the other a *little* system. The terms "big" and "little" are used in [Zei76]. These terms should be interpreted in contrast to each other. That is, a big system should be understood to have more input, output, and/or state variables as well as more elaborate transition functions than a little system. For example, a big system may be a "real" system, whereas the little system could be a "model" of the real system. A theory of abstraction attempts to establish some formal means to examine the behavioral similarities of these systems. In particular, it is desirable to establish the correctness of a little system with respect to the big system.

Any formalization of abstraction should be carried out in terms of the level of model specification in the hierarchy of system specifications (refer to Section 2.3). Abstraction can be categorized into two types. First, abstraction can be used at any one level in the system's hierarchy, that is, an abstraction of an IO system (i.e., big system S_1) is another IO system (i.e., little system S_2). The other type of abstraction deals with one system at level i and another at level j, where $0 \leq i, j \leq 5$ and $i \neq j$. For instance, an IO system model ($j = 2$) can be abstracted as an IO function observation ($i = 1$).

Before any further discussion of reasoning and abstraction in the context of inductive modeling can be attempted, a brief account of well-studied abstractions in deductive modeling will be helpful.

Within the deductive modeling paradigm, abstraction has been formalized using

Systems Theory [Zei76, Zei84, Wym93] and Logic [CK90, GW92]. Using the system specification hierarchy given in Section 2.3, I view system specifications based on levels 0-2 and levels 3-5 as inductive and deductive modeling paradigms, respectively. I begin with a short discussion of abstraction in the context of deductive modeling. First, I give a definition of abstraction derived from [GW92]:

Definition 12 *An abstraction is a pair of distinct formal systems and a mapping between them.*

Note that, according to this definition, abstraction establishes a relationship between two systems. However, the definition does not specify how to actually go about abstracting one from the other. Both systems theory and logic have been used in developing theories of abstractions. I start with the systems theory view of abstraction.

Generally, abstraction is understood to be the simplification of one model into another via *homomorphism* (a preservation relation) [Sto73, Zei76, Wym93]. Both Wymore and Zeigler use homomorphism as the *basis* for abstraction. For instance, at the I/O system specification level, given a big system $S_1 = \langle T, X, \Omega, Q, Y, \delta, \lambda \rangle$ and a little system $S_2 = \langle T', X', \Omega', Q', Y', \delta', \lambda' \rangle$, then S_2 is an abstraction of S_1 if some aspect of the big system structural specification is preserved in S_2. That is, a *system morphism* at the I/O System level provides *checkable* conditions such that some structural characteristics of S_1 are preserved in S_2. Therefore, some desired behavioral patterns of the big system should be derivable from the little system.

The I/O system morphism from S_1 to S_2 is defined as a set of mappings for inputs, outputs, and states as well as transition and output preservation functions (for a detailed exposition refer to [Zei76]):

Definition 13 *A system morphism from $S_1 = \langle T, X, \Omega, Q, Y, \delta, \lambda \rangle$ to $S_2 = \langle T', X', \Omega', Q', Y', \delta', \lambda' \rangle$ is a triple (g, h, k) such that [Zei76]:*

1. $g : \Omega' \longrightarrow \Omega$

2. $h : \widehat{Q} \xrightarrow{onto} Q'$ *where* $\widehat{Q} \subseteq Q$
3. $k : Y \xrightarrow{onto} Y'$
4. $h(\delta(q, g(\omega'))) = \delta'(h(q), \omega')$
5. $k(\lambda(q)) = \lambda'(h(q))$

where $q \in \widehat{Q}$ *and* $\omega' \in \Omega'$.

I need to discuss abstraction from the viewpoint of logic as well since the reasoning part of my work is based on the tenet of logic. Although abstraction has been studied in systems theory and logic separately, the basic underlying principles are the same in both cases. Nevertheless, the theory of abstraction based on logic is richer due to the inherent power of the language of logic.

In logic, a system is a collection of *wffs* (theorems and axioms), and it is called a *theory.* Therefore, in contrasting two systems, we need to consider two theories, one representing the big system and another the little system. In logic, however, a system has to be discussed in terms of its interpretations (i.e., models). That is, we use interpretations in order to contrast two systems (i.e., their theories) in terms of one being an abstract representation of the other.

Given two axiomatic formal systems $\langle \Sigma_1, \Sigma_2 \rangle$, where $\Sigma_i = \langle \Lambda_i, \Delta_i, \Omega_i \rangle$, Λ_i a language, Ω_i a set of axioms, and Δ_i a deductive machinery (inference rules), we have the following definitions:

Definition 14 *An abstraction, written* $f : \Sigma_1 \longrightarrow \Sigma_2$*, is a pair of axiomatic formal systems* $\langle \Sigma_1, \Sigma_2 \rangle$*, and a function* $f_\Lambda : \Lambda_1 \longrightarrow \Lambda_2$ *[GW92].*

Given two theories, $\mathcal{T}(\Sigma_1)$ (the concrete theory) and $\mathcal{T}(\Sigma_2)$ (the abstract theory), we have:

Definition 15 *An abstraction* $f : \Sigma_1 \longrightarrow \Sigma_2$ *is said to be a [GW92]:*

1. *TD-abstraction iff, for any wff* $\phi \in \mathcal{T}(\Sigma_1)$*, if* $\varphi = f_\Lambda(\phi) \in \mathcal{T}(\Sigma_2)$*, then* $\phi \in \mathcal{T}(\Sigma_1)$*,*

2. *TI-abstraction iff, for any wff* $\phi \in \mathcal{T}(\Sigma_1)$*, if* $\phi \in \mathcal{T}(\Sigma_1)$*, then* $\varphi = f_\Lambda(\phi) \in \mathcal{T}(\Sigma_2)$.

These two definitions are called TI-abstraction (theorem-increasing abstraction) and TD-abstraction (theorem-decreasing abstraction). Both types of abstractions indicate that $\mathcal{T}(\Sigma_2)$ is more abstract than $\mathcal{T}(\Sigma_1)$. The TI-abstraction ensures that the absence of a theorem in the abstract theory implies its absence in the concrete theory as well. Here, some of the abstract theory's conclusions may not be supported by the concrete theory. In contrast, the TD-abstraction ensures that all conclusions made by the abstract theory are also correct in the concrete theory. In other words, a TI-abstraction allows more conclusions, some of which may be wrong, whereas a TD-abstraction allows fewer conclusions, but all are correct. My brief discussion leaves many important things untold. For a comprehensive exposition of a theory of abstraction based on logic, accompanied with numerous examples, refer to [GW92].

Note that the notion of abstraction in logic is somewhat different than the one defined for systems theory. Here, mapping is defined over a "language," instead of mapping the input, output, and state sets as is done in systems theory.

Having defined two formulations of abstraction in systems theory and logic, we can observe that both of these provide some means to evaluate relationships between a system and any of its abstractions. However, neither of them provides a *procedure* that would specify how an abstraction of a system is actually obtained. For instance, given some input set X in S_1, Definition 13 does not tell us how to obtain X' in S_2. Likewise, given a theorem $\phi \in \mathcal{T}(\Sigma_1)$, Definitions 14 and 15 provide no means to obtain a theorem $\varphi \in \mathcal{T}(\Sigma_2)$.

Now I turn to abstraction at the input/output level specification. As stated in Chapter 2, an IOFO specification is an algebraic structure with input, output, and state sets, an input/output space, and an input/output function set.

Again, the basis for abstraction is the notion of homomorphism between $IOFO_1$ (a big system) and $IOFO_2$ (a little system) that can be used to ensure the preservation of some input/output behavioral patterns of the big system in the little system. An I/O function observation morphism for the IOFO level is:

Definition 16 *Given S_1 ($IOFO = \langle T, X, S_i, Y, IOspace, F \rangle$) and S_2 ($IOFO' = \langle T', X', S_i', Y', IOspace', F' \rangle$), an I/O function morphism from S_1 to S_2 is a pair (g, k) such that [Zei76]:*

1. *$g : \Omega' \rightarrow \Omega$*
2. *$k : \Psi \xrightarrow{onto} \Psi'$*
3. *For each $f' \in F'$, there is an $f \in F$ such that for all $\omega' \in \Omega', f'(\omega') = k(f(g(\omega')))$; that is $f' = k \circ f \circ g$.*

Items (1) and (2) are self-explanatory. Item (3) is saying that if S_2 is in a state whose I/O function is f' and an input history ω' is injected into it, then ψ' can be observed. Now, using the I/O function morphism mappings, we can observe the same output history ψ' for ω' by first mapping ω' into ω using function g. Then, the output history ψ for S_1 can be obtained using f, which can subsequently be mapped into ψ' using k. These mapping functions can be used to verify the correspondence between S_1 and S_2 (in terms of their behavior preservation). Again, the I/O function morphism is used for verification purposes. It is not a procedure for deriving the mappings g and k.

Although the above concepts and definitions are quite useful, primarily in deductive modeling methodologies, they are not suitable for our purposes. This is mainly due to the absence of any procedure that can derive, for example, S_2 from S_1 in Definition 5. Appropriate settings should be developed in order to derive such algorithms.

5.2 IO Segment Pair Types

Before I proceed, we may recall that the I/O function generator set F_G is simply a database containing pairs, each comprised of an initial state associated with its input/output segment pair. That is,

$$F_G : S_i \longrightarrow \text{partial } IOspace_G,$$

$$f = (s, g) \in F_G \qquad \text{where} \qquad g = (\omega, \psi).$$

In this form, no final state is assigned to any IO segment pair. Instead, the quasi-state identification function γ_G is specified in G_F to hypothesize about them. For any particular set of final states, I can suppose that every IO segment has both an initial state s_i and a final state s_f (i.e., $(s_i, s_f, (\omega, \psi))$). It should be noted that $s_i, s_f \in S_i$ (see Section 3.2).

My discussion of trajectories and segments in Chapter 2 did not consider how IO trajectories may be partitioned into IO segments. In addition, it did not discuss the possible ways in which input segments, output segments, and pairs of input/output segments may be represented. In this section, we begin with a classification of all possible ways in which an input segment might be represented.

Let me denote an input segment as (s_i, ω), an output segment as (s_f, ψ), and a causal IO segment pair as $((s_i, \omega), (s_f, \psi))$. An input segment's representation, without considering its initial and final states for now, can be categorized into several types depending on whether events occur at one or both of its initial and final time-points; likewise, for output segments. For example, suppose we have an input segment ω and an output segment ψ with duration dt. We associate t_i with the initial time-point of the input/output segment pair and t_f with its final time-point. As has been said previously, we are concerned with discrete-event systems only. Hence, all segments are assumed to be of the discrete-event type. Thus, we can have the following discrete-event input segment types:

- Input segment type 1:

$$\omega(t) = nil \quad \text{for} \quad t_i \leq t \leq t_f$$

- Input segment type 2:

$$\omega(t) = \begin{cases} input\_event & \text{for} \quad t = t_i \\ nil & \text{for} \quad t_i < t \leq t_f \end{cases}$$

- Input segment type 3:

$$\omega(t) = \begin{cases} nil & \text{for} \quad t_i \leq t < t_f \\ input\_event & \text{for} \quad t = t_f \end{cases}$$

- Input segment type 4:

$$\omega(t) = \begin{cases} input\_event & \text{for} \quad t = t_i \\ nil & \text{for} \quad t_i < t < t_f \\ input\_event & \text{for} \quad t = t_f \end{cases}$$

Likewise, output segments are of the same four types. Then, considering input and output segments together, I require that each conform to one of the following 4 types out of all 16 possible combinations of IO segment pairs.

- IO segment type 1:

$$\omega(t) = nil \quad \text{for} \quad t_i \leq t \leq t_f$$

$$\psi(t) = nil \quad \text{for} \quad t_i \leq t \leq t_f$$

- IO segment type 2:

$$\omega(t) = \begin{cases} \mathit{input\_event} & \text{for} \quad t = t_i \\ \mathit{nil} & \text{for} \quad t_i < t \leq t_f \end{cases}$$

$$\psi(t) = \mathit{nil} \qquad \text{for} \qquad t_i \leq t \leq t_f$$

- IO segment type 3:

$$\omega(t) = \mathit{nil} \qquad \text{for} \qquad t_i \leq t \leq t_f$$

$$\psi(t) = \begin{cases} \mathit{nil} & \text{for} \quad t_i \leq t < t_f \\ \mathit{output\_event} & \text{for} \quad t = t_f \end{cases}$$

- IO segment type 4:

$$\omega(t) = \begin{cases} \mathit{input\_event} & \text{for} \quad t = t_i \\ \mathit{nil} & \text{for} \quad t_i < t \leq t_f \end{cases}$$

$$\psi(t) = \begin{cases} \mathit{nil} & \text{for} \quad t_i \leq t < t_f \\ \mathit{output\_event} & \text{for} \quad t = t_f \end{cases}$$

The rationale for this restriction is the following: The trajectories for a single input, single output (SISO) system, can be partitioned in several ways. Clearly, the partitioning of IO trajectories for an SISO system ought to be with respect to both input and output trajectories. If there are no time-points at which both an input event and output event occur, then the partitioning of IO trajectories results in IO segments having the above proper types. That is, no input event and output event should occur at the same time in order to ensure that an IO segment pair has one of the proper types defined above. A trajectory may be partitioned at time-points where either an input event or an output event occurs. Thus, partition points (time instants along a trajectory where an event occurs) should occur only at time-points where either an input event or an output event occurs, exclusively.

In particular, the following types of input and output segments are not allowed due to violation of the causality principle.

$$\omega(t) = \begin{cases} input\_event \text{ or } nil & \text{for} \quad t = t_i \\ nil & for \quad t_i < t < t_f \\ input\_event & \text{for} \quad t = t_f \end{cases}$$

$$\psi(t) = \begin{cases} output\_event & \text{for} \quad t = t_i \\ nil & for \quad t_i < t < t_f \\ output\_event \text{ or } nil & \text{for} \quad t = t_f. \end{cases}$$

Therefore, partitioned IO segment pairs should adhere to the 4 types described above. In particular, input segments of types 3 and 4 cannot be used. Figure 5.1 illustrates pictorially an input/output trajectory and its partitioned IO segments.

Having decided on the proper representations for input and output segments, I can now determine the representation of *Complete IO segments*[1]. That is, any IO trajectory pair (partitioned according to assumption set-I) would result in IO segment pairs representable as:

$$((s_i, (x_{val}, dt)), \quad (s_f, (y_{val}, dt)))$$

where dt is the duration of input/output segments and $x_{val} \in \{nil, input\_event\}$ and $y_{val} \in \{nil, output\_event\}$. Note that the durations of the input and output segments of an input/output pair must always be identical. Hence I can reformulate the earlier specification of G_F in terms of $((s_i, (x_{val}, dt)), \ (s_f, (y_{val}, dt)))$ as:

$$\widehat{G_F} = \langle T, X, S, Y, IOspace_G, \widehat{F_G} \rangle \qquad \text{where}$$

[1] The prefix Complete, which distinguishes an IO segment with states from an IO segment without states, is dropped as long as there is no confusion.

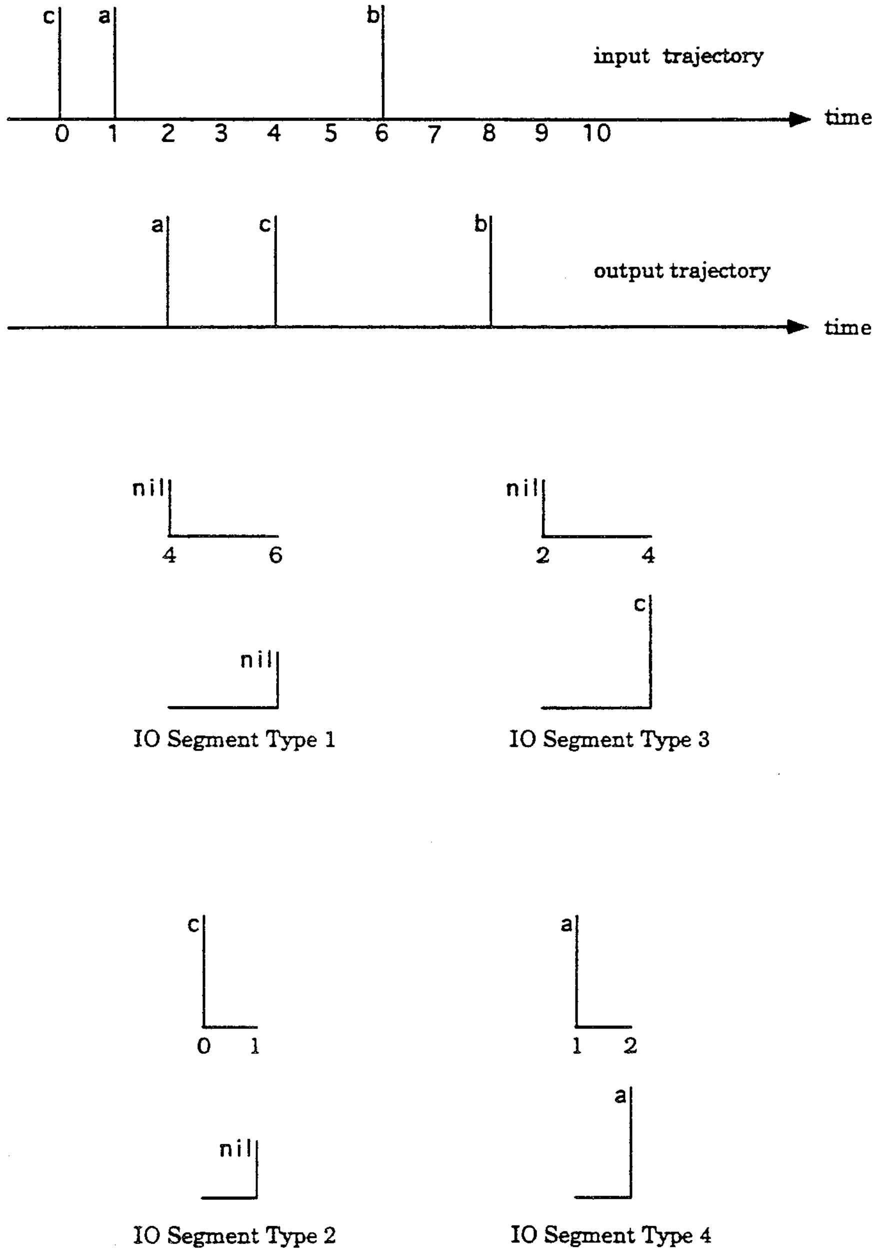


Figure 5.1, A discrete-event input/output trajectory and its IO segments

T	time base
X	input value set
Y	output value set
S	set of states
$IOspace_G$	time-invariant IO segment generator set
$\widehat{F_G}$	partial IO function generator set

and

$$\begin{array}{rl} \widehat{F_G}: & S_i \times PJN(IOspace_G, 1) \longrightarrow S_f \times PJN(IOspace_G, 2) \qquad \text{such that} \\ & S_i \subseteq S, S_f \subseteq S. \\ \widehat{F_G} = & \{((s_i, (x_{val}, dt)),\ (s_f, (y_{val}, dt)))\ \mid \quad s_i, s_f \in S, \\ & (x_{val}, dt) \in PJN(IOspace_G, 1), \quad \text{and} \\ & (y_{val}, dt) \in PJN(IOspace_G, 2)\}. \end{array}$$

Therefore, $\widehat{G_F}$ represents either a set of observed IO segments, each having associated with it an initial state and a final state, or an iterative IOFO specification.

5.3 Equivalence Relations

The main idea behind the iterative IOFO specification is to provide predictability. Partitioning trajectories into segments allows a combinatorial composition of segments into trajectories. The IO segments may be observed or be the result of partitioning IO trajectories. In general, partitioned IO trajectories based on assumption set-I are subject to revision. It is possible however, that after a finite number of revisions, IO segments need not be revised any further. That is, whereas F_G may be subject to revision, $\widehat{F_G}$ is not. I shall work with $\widehat{F_G}$ in the remainder of this work.

There is no fundamental difficulty with my supposition. It is common practice in inductive as well as deductive qualitative modeling to abstract inputs, outputs, and states. The difference (advantage) in treating a data set, derived according to assumption set-I, is that reasoning with such abstractions is possible in ways similar to those a human modeler would use. Therefore, I neither consider nor develop the means by which the assumption set-I can be used to derive an iterative IOFO specification.

The assumption set-II is to be used to reason about IO segments contained in $\widehat{F_G}$ and a candidate input segment to predict unobserved IO segments. An input segment is called a *candidate* input segment when it is not observed. A *concrete* input segment (or IO segment), however, refers to one that is observed. We need to be specific about what the assumption set-II is, and what it entails given $\widehat{G_F}$.

Suppose we are given an input trajectory partitioned into a finite number of sequential segments. Then, we are interested in finding its corresponding output trajectory. For example, suppose one of its input segments is $(s'_i, (x'_{val}, dt'))$, and there exists a pair of input/output segment $((s_i, (x_{val}, dt)),\ (s_f, (y_{val}, dt))) \in \widehat{F_G}$. If the input segment $(s'_i, (x'_{val}, dt')$ is *equal* to the input segment $(s_i, (x_{val}, dt))$ (i.e.,

$x_{val} = x'_{val}$, $dt = dt'$, and $s_i = s'_i$), then we are able to postulate the output segment corresponding to this input segment by analogy.

However, the hope of composing the output trajectory is dashed once and for all if, for a given input segment, no equal input segment be found in the database. For instance, given the three sequential input segments $(s'_1, (x'_1, dt'_1))$, $(s'_2, (x'_2, dt'_2))$, and $(s'_3, (x'_3, dt'_3))$, it is entirely irrelevant whether output segments can be found for the second or third input segment, as long as no output segment can be found for the first input segment. That is, given the candidate input segment $(s'_1, (x'_1, dt'_1)) \notin PJN(\widehat{F_G}, 1)$, (i.e., there exists no input/output segment pair $((s_i, (x_{val}, dt)),\ (s_f, (y_{val}, dt))) \in \widehat{F_G}$ in the database, such that $x_{val} = x'_1$, $dt = dt'_1$, and $s_i = s'_1$), then no output trajectory can be obtained.

The notion of equality, of course, is *too strong* for inductive modeling. It becomes imperative to speak of *equivalence* instead[2]. Otherwise, we have to limit our claims of prediction to trajectories that can be composed from the IO segments found in $\widehat{F_G}$ only. Without using equivalence, no generalization capability would be present in the reasoner, and matching of input segments would be limited to what is available in $\widehat{F_G}$. It is the inevitable impossibility to find in the database equal input segments for all imaginable new input segments (due to the qualification problem) that underlie the need for an equivalence definition[3]. I use the term equivalence to indicate that, even though two input segments are not equal, we can *think* of them as being equal.

When are two segments said to be "equivalent" instead of equal? An equivalence relation is weaker than the equality relation. Therefore, it is necessary to decide,

[2] Our usage of the term equivalence is different from the one used in [Gil62] where various notions of (deductive) equivalence are defined for finite-state memory machines. For instance, a set of states for a deductive model is shown to have an equivalent set of states that is smaller. Then, a homomorphism relation is used to show that the system has the same behavior for either of the two sets of equivalent states.

[3] The type of equivalence we are seeking is not based on fuzzy logic or probability theory.

given two input segments $(s_i^{'}, \omega^{'}) \notin PJN(\widehat{F_G}, 1)$ and $(s_i, \omega) \in PJN(\widehat{F_G}, 1)$, when they can be *considered to be equal*, i.e., when they can be called *equivalent.* Two input segments, $(s_i, (x_{val}, dt))$ and $(s_i^{'}, (x_{val}^{'}, dt^{'}))$, are equal iff $x_{val} = x_{val}^{'}$, $s_i = s_i^{'}$, and $dt = dt^{'}$. Given the two input segments $(s_i, (x_{val}, dt))$ and $(s_i^{'}, (x_{val}^{'}, dt^{'}))$, three primitive types of equivalence relations are possible. They are based on *length-equivalence*, *input-equivalence*, and *state-equivalence.* Each of these equivalence definitions discards the inequality in one of the three aspects: length (or duration), initial state, or input. Hence we have the following definition:

Definition 17 *Two input segments $(s_i, (x_{val}, dt))$ and $(s_i^{'}, (x_{val}^{'}, dt^{'}))$ are called:*

1. *length-equivalent iff*

$$\begin{aligned} x_{val} &= x_{val}^{'}, \\ dt &\neq dt^{'}, \\ s_i &= s_i^{'}; \end{aligned}$$

2. *input-equivalent iff*

$$\begin{aligned} x_{val} &\neq x_{val}^{'}, \\ dt &= dt^{'}, \\ s_i &= s_i^{'}; \textit{ and} \end{aligned}$$

3. *state-equivalent iff*

$$\begin{aligned} x_{val} &= x_{val}^{'}, \\ dt &= dt^{'}, \\ s_i &\neq s_i^{'}. \end{aligned}$$

that is, despite the presence of an inequality in each of the above equivalence relations, the two input segments are *believed* to be equivalent. An equivalence relation between two input segments violates exactly one of the three equalities relating to input, duration, and state, that are mandated by the equality relation. Hence, given two input segments, either the *length*, the *input*, or the *state* can be ignored in an equivalence relation. Let us call these `abs-length`, `abs-input`, and `abs-state` for future use in Chapter 6. Any combination of these comprises one form of assumption set-II. A more specific form of assumption set-II is discussed in Section 5.4.4.

5.4 Discrete-event Inductive Reasoner

In the previous sections, it has been convenient not to be concerned about *truth* in my discussions. In studying the $\widehat{G_F}$ structure, everything was considered to be true, which relieved us from considering truth as being negotiable. Not having to ponder about the truth of a statement facilitates the manipulation of the the iterative IOFO structure. Deciding about the equality of two input segments requires no explicit comparison of their truth values as long as the equality relation is satisfied. What happens if two input segments do not satisfy the equality relation, but some of their parts do? I already discussed cases where, for instance, two input segments were equal in their inputs and durations, but not equal in their state values. I pointed out that we must free ourselves from the rigidity of the equality relation in order to tackle the qualification problem. I did not, however, discuss what is necessary to warrant the use of the previously proposed equivalence relations. We must be careful in ensuring the proper use of equivalence relations.

In this section, I discuss how NMR provides well-defined mechanisms supporting the use of equivalence relations in place of the equality relation when applicable. Specifically, I consider how the iterative IOFO specification $\widehat{G_F}$ can be augmented with an inference engine and a truth maintenance system to create a more general and versatile IOFO structure. I shall refer to this problem solver as a *Discrete-event Inductive Reasoner (DIR)* (see Figure 5.2.)

The truth maintenance system of DIR carries out the tasks discussed in Chapter 4. It consists of a database (i.e., observed and predicted IO segments) and an inference engine responsible for maintaining consistency among its databases. DIR also has its own inference engine which is distinct from the above inference engine. One of the roles of the DIR's inference engine is to predict (infer) unobserved IO segments. It relies on equivalence relations (i.e., abstraction mechanisms and

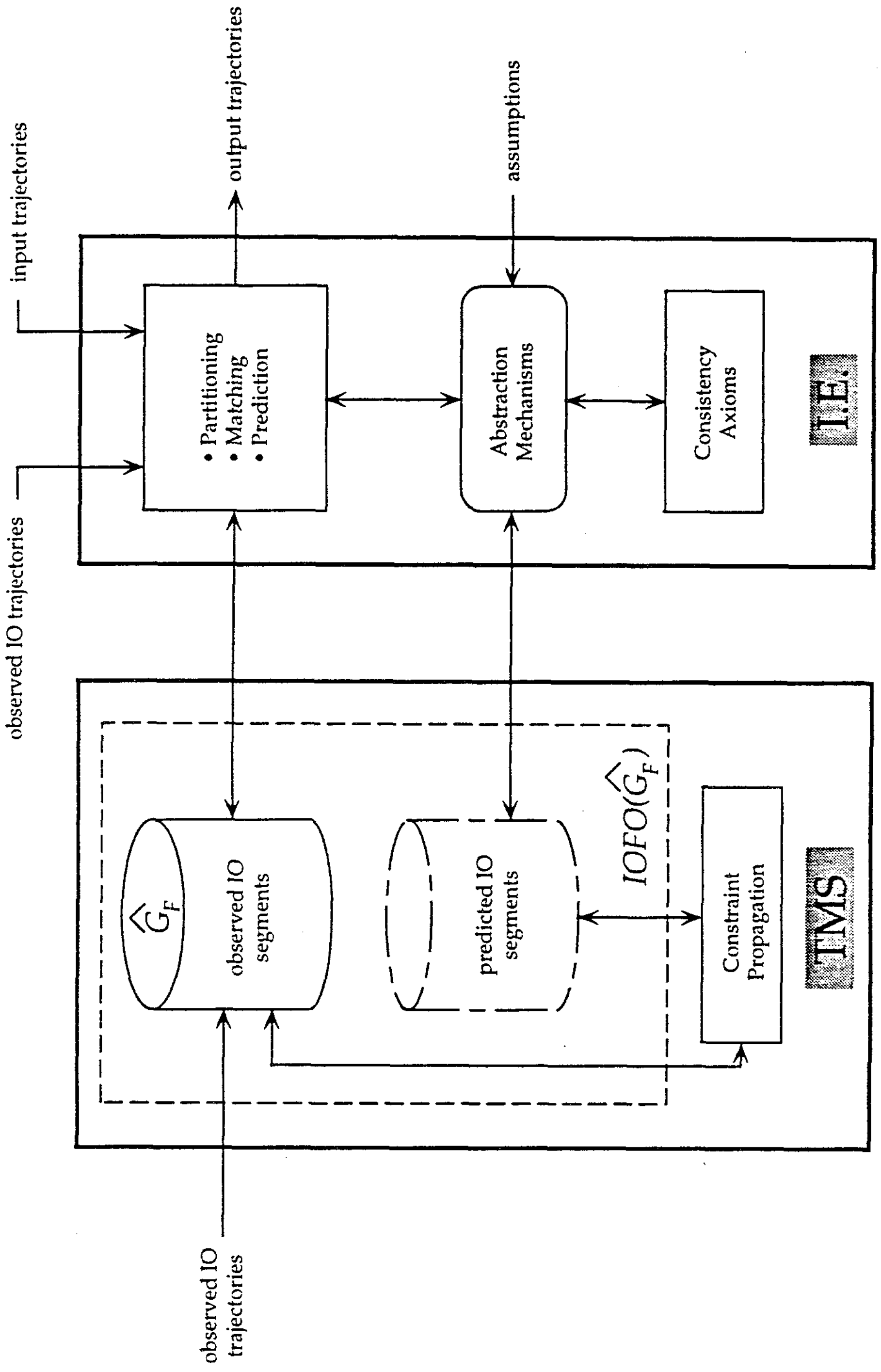


Figure 5.2, Discrete-event Inductive Reasoner

consistency axioms to be developed in the section) to overcome the restrictions of the iterative IOFO structure described in Section 5.2. The inference engine of DIR also has the responsibility of matching any two IO segments against each other as well as partitioning an IO trajectory into IO segments. The TMS supports proper use of those IO segments that are supported by the equivalence relations.

The additional capabilities of this new structure are due to the equivalence relations and the non-monotonic reasoning capabilities offered by the TMS. The resultant structure is no longer restricted to the observed data — it can make "sound" tentative predictions about IO segments.

I have already discussed in detail the iterative IOFO structure. I also discussed the equivalence relations and their properties. What I have not done yet is to show how these three basic elements (i.e., the iterative IOFO specification, the equivalence relations, and a TMS) can work together to form the discrete-event inductive reasoner.

It is useful to pose some questions about the DIR. Thus far, in various settings, I have discussed primarily the why and what questions. I have argued for the need of DIR. I also discussed, in general terms, what the tasks of DIR are. In comparison, I have said little about how the DIR goes about predicting an output segment for an input segment. However, before delving into any details about the specifics of how DIR works, I shall give an overview of the fundamental properties of the Discrete-event Inductive Reasoner.

In the previous chapter, I discussed a problem solver that had an inference engine and an LTMS as its basic elements. The fundamental motivation for such a problem solver was the lack of complete knowledge about the problems that were to be solved by it. The inference engine would make decisions where some of them would be tentative. The maintenance of a consistent set of data was the responsibility of the TMS. The inference engine and the TMS, therefore, had to communicate with each other such that their activities could complement each other.

My earlier reference to the DIR as a problem solver was no accident! It is in fact a problem solver. A truth maintenance system and an inference engine as depicted in Figure 5.2 together can infer new input/output segments from observed and predicted IO segments. Now, what kind of TMS can serve the purposes of DIR? Each equivalence relation is essentially used to hypothesize about an unobserved IO segment. Since it is more general to assign TRUE, FALSE, and UNKNOWN truth values to IO segments, I have chosen a logic-based TMS. Figure 5.3 shows the basic architecture of the *Logic-based Discrete-event Inductive Reasoner.* Hence, LDIR relies on LTMS to maintain a consistent set of IO segments, some of which are predicted.

Therefore, we can identify DIR as a problem solver where its inference engine is the machinery of the iterative IOFO structure along with the equivalence relations. The inference engine either finds output segments for a candidate input segment that match the input segment of an IO segment available in $\widehat{F_G}$, or it hypothesizes an input/output segment for a candidate input segment that is not available in $\widehat{F_G}$, but that is consistent with other existing IO segments. Whenever a hypothesized IO segment or observed IO segment becomes available, each is added to the existing propositions in the LTMS while maintaining consistency among all of them. That is, each IO segment pair has a truth value assigned to it, and the LTMS only adds those IO segments that are consistent with what it already knows to be true.

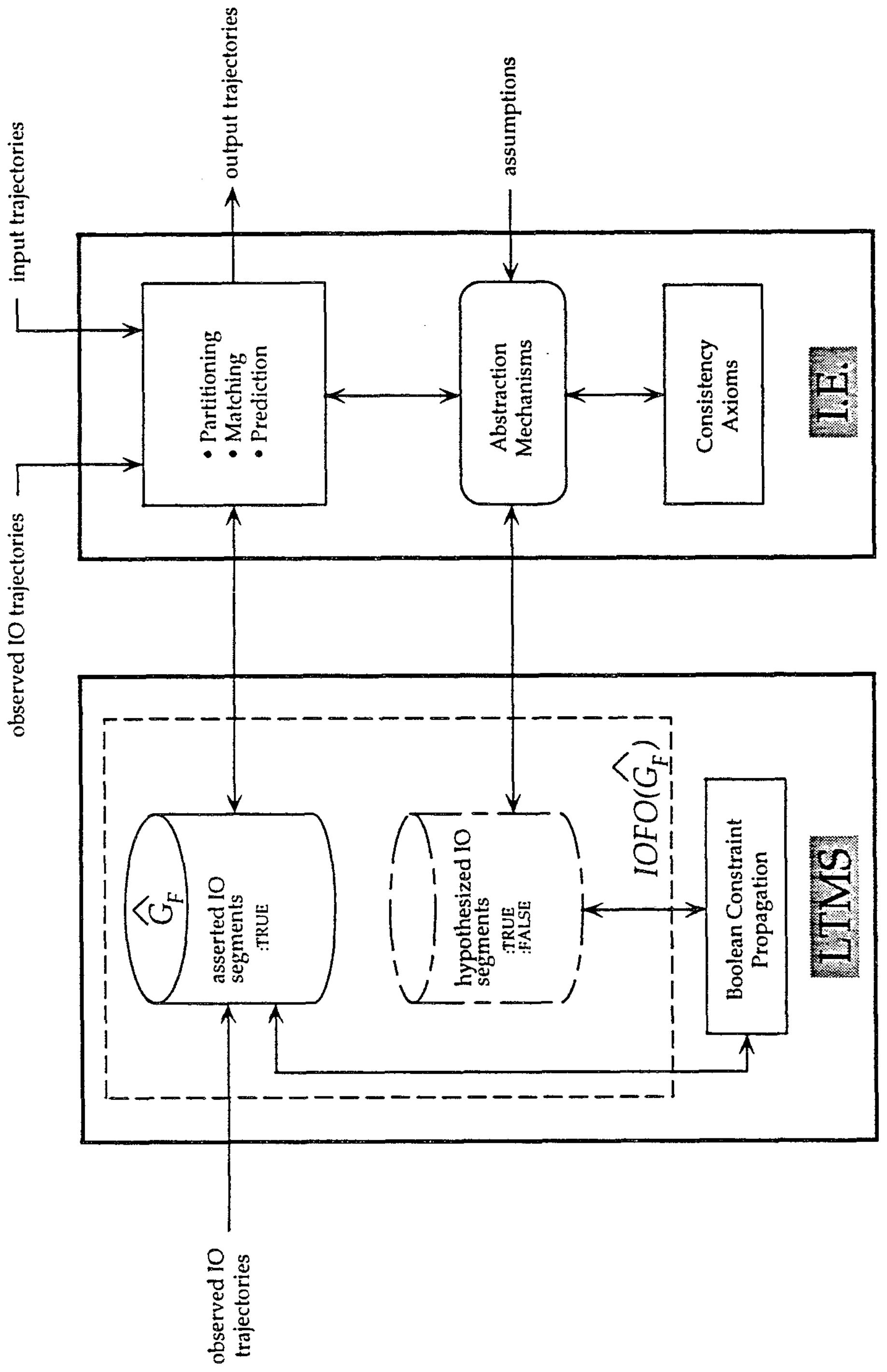


Figure 5.3, Logic-based Discrete-event Inductive Reasoner

5.4.1 Representation of IO Segments

I begin with knowledge representation of IO segments for the inference engine and the LTMS. I illustrate by an example. A DIR is expected to compute output segments for unobserved candidate input segments. Let us suppose the iterative IOFO contains an IO segment

$$((s_i, (x_{val}, dt)), \quad (s_f, (y_{val}, dt))) \in \widehat{F_G}.$$

Then, the DIR receives a candidate input segment

$$(s_i', (x_{val}, dt)) \notin PJN(\widehat{F_G}, 1)$$

for which it is expected to compute the following IO segment based on the existing IO segment and the state-equivalence relation.

$$((s_i', (x_{val}, dt)), \quad (s_f, (y_{val}, dt)))$$

Starting at the lowest level, we need to decide about the language in which knowledge (e.g., IO segments) should be represented for the inference engine and the LTMS. In my discussions of the problem solver, I said that the available data to the inference engine need to be presented in the logic-based TMS as well. Of course, the type of knowledge representation for the inference engine and the LTMS need not be the same!

In the previous chapter, I argued for a logic-based TMS. Hence we would have to encode IO segments and other necessary data (e.g., justifications) in the language of propositional logic. However, I do not use the language of propositional logic for the representation of IO segments in the iterative IOFO structure. The iterative IOFO structure is basically a database containing the observed IO segments along with some functions allowing it to "match" two input segments from which a proper output is chosen. There is no need to use an expensive Theorem Prover, which would require a logical language for knowledge representation and an inferencing procedure such as resolution principle. One can, of course, use a Theorem Prover if this should be necessary.

I already discussed how IO segments of an iterative IOFO should be encoded. Each input or output segment can be represented as a list. Then, I can represent an IO segment as a nested list containing the list for the input segment and the list for the output segment.

Next, I need to decide how an IO segment such as $((s_i, (x_{val}, dt)),\ \ (s_f, (y_{val}, dt))$ is to be represented in logic. There are several choices depending on the level of aggregation. In the least aggregated representation, the state value, input value, and duration value of a segment are considered distinct non-logical symbols. We can represent the initial state, s_i, the input, x_{val}, and the duration, dt, of an input segment as three separate non-logical symbols `si`, `a`, and `dt` respectively. With this choice, I am claiming that there are objects in the universe of discourse that correspond to them. Using these non-logical symbols together with the logical symbol $\wedge$, I can form a *wff* such as

$$\texttt{si} \wedge \texttt{a} \wedge \texttt{dt}$$

to represent an input segment. Having a representation for an input segment, I can use the same representation for an output segment with the final state `sf` and the output value `b`:

$$\texttt{sf} \wedge \texttt{b} \wedge \texttt{dt}$$

Another choice is to represent an entire input segment as one non-logical symbol. That is, the k^{th} input segment $(s_i, (x_{val}, dt))$, represented as **in.seg.k**, corresponds to the k^{th} input segment in the universe of discourse. Then, using the same representation scheme for an output segment, we can form *wffs* relating an input segment to an output segment like:

$$\texttt{in.seg.k} \Longrightarrow \texttt{out.seg.k}$$

or

$$\texttt{in.seg.k} \wedge \texttt{out.seg.k}.$$

The most aggregated representation corresponds to denoting a pair of I/O segments as a single non-logical symbol having a corresponding counterpart in the universe of discourse. For instance, the k^{th} IO segment $((s_i, (x_{val}, dt)), \ (s_f, (y_{val}, dt)))$ can be represented as

$$\texttt{io.seg.k}.$$

In this form, logical symbols are not used to indicate, for instance, that an output segment and an input segment are related in a specific way. There is no explicit relationship between input and output segments. If both non-logical symbols in in.seg.k$\wedge$out.seg.k are assigned the truth value TRUE (or FALSE), then there is no difference between io.seg.K and in.seg.k $\wedge$ out.seg.k. However, the assignment of different truth values to the input and output segments makes the conjunction subject to more interpretations (i.e., truth assignments). Whereas the input and output segments can be used independently to reason about the truth of their conjunction in the less aggregated representation, this is not possible in the more aggregated representation.

Each level of aggregation has its own semantics. For instance, whereas the least aggregated representation allows reasoning with each of the elements of a segment, the most aggregated representation allows reasoning about an IO segment as a whole. The least aggregated form might appear to be the best choice since reasoning can be focused on elements of each IO segment. With this choice, however, we are forced to delve into ontological issues about state, input value, and duration of a segment as well as the relationship between the input and output segments. In considering the relationship between the input and output segments, we have to use either a conjunction or an implication primitive to relate an input segment to an output segment. Each enforces a particular semantics on the relationship between them. The truth assignments to these two sentences are completely different. Consequently, it is important to choose one that reflects the intended truth about an IO segment pair.

In order to come to a conclusion about this issue, I can ask which representation is the simplest scheme that allows prediction as well as revocation of input/output segments. Clearly, the most aggregated representation lends itself to more efficient reasoning. I am faced with the question of whether or not the most highly aggregated representation of the IO segment can support the prediction and revision of IO segments. Given the degree of reasoning we are interested in, the answer is affirmative. Thus, I use the most aggregated representation `io.seg.k` in the remainder of this work.

5.4.2 Authorizing the Use of Equivalence Relations

Earlier in this chapter, I showed different types of equivalence relations between two input segments. I now need to discuss what is required to authorize their use. In the description of the equivalence relations, I referred to the term "belief." Also I have been using the term "fact." These two terms are generally used without much attention to their semantics. They are related, but are two distinct concepts, especially in the present setting.

I use the term *fact* in the restricted sense that its truth value is fixed and cannot be subjected to revision. The term *belief*, however, can change its truth value. Another difference between these is that, whereas a belief can be converted to a fact, the converse is not true. Every piece of data is either a fact or a belief, exclusively. In my earlier reference to a database, as used in relation to the DIR, I said that it consists of facts and beliefs. This type of database is needed for systems expected to operate with partial knowledge. I use the term partial knowledge to refer to a database consisting of facts and beliefs.

A candidate input segment is not observed: thus, it should be treated as a belief rather than as a fact. What makes a predicted IO segment a belief and not a fact? I said that, if an IO segment is not observed, it is not a fact. Then, since each IO segment is either a fact or a belief, a predicted IO segment has to be a belief.

The concept of belief is central in using equivalence relations. It is important to understand the interplay between belief and equivalence relations. I proceed by looking at how hypothesized IO segments come to be actually identified as beliefs. We know that one or more of the equivalence relations must be used to hypothesize an IO segment. We also know that the equivalence relations were defined in terms of input segments rather than IO segments.

I start with discussing how an equivalence relation can be justified. Even though I argued already in the previous section for not assigning truth values to input segments, we want to think of them for the moment as having truth values.

Let us think of an entity as an input segment, an output segment, an IO segment, or an equivalence relation. An entity that represents a belief can be assigned any of the three truth values TRUE, FALSE, or UNKNOWN, just as a fact can be assigned these truth values. If a belief is said to be true, then it is assigned the truth value TRUE. Likewise, if it is said to be false, then its assigned truth value is FALSE. If a belief's truth value is neither TRUE nor FALSE, then its assigned truth value is UNKNOWN. Conjunctions of beliefs and facts are also beliefs since the

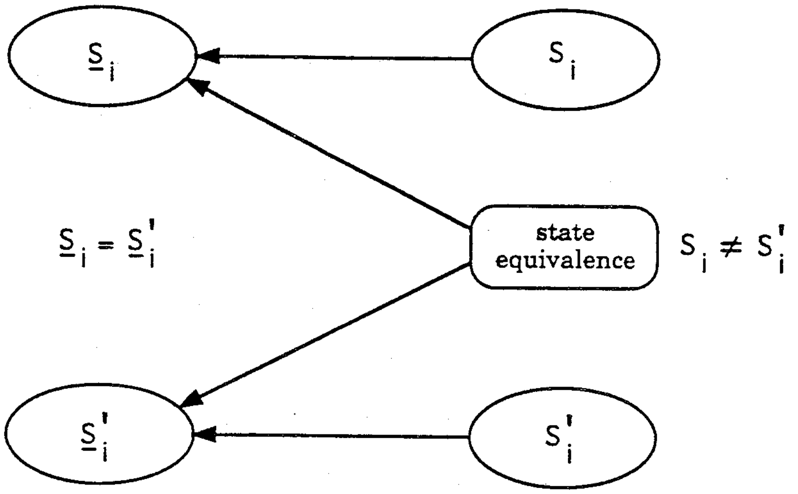


Figure 5.4, Abstraction of two concrete states into their respective abstracted states

truth assignment of the conjunction depends on the truth values assigned to the individual beliefs.

In my earlier example, we had $(s_i, (x_{val}, dt))$ and $(s_i', (x_{val}', dt'))$, where $s_i' \neq s_i$, $x_{val}' = x_{val}$, and $dt' = dt$. Consequently, we need to assume state-equivalence in order to consider these two segments as being equivalent. How can this formally be accomplished? One approach is to turn the inequality $s_i' \neq s_i$ to an equality between their respective abstracted states $\underline{s}_i$ and $\underline{s}_i'$. Now, I define the state-equivalence between s_i and s_i' as an *assumption.* Then, we can use $s_i' \neq s_i$ (fact) along with the state-equivalence assumption (belief) to construct two abstract states $\underline{s}_i$ and $\underline{s}_i'$ from the concrete states s_i and s_i' (cf. Figure 5.4), where

$$\underline{s}_i' = \underline{s}_i.$$

Hence the use of the state-equivalence relation is substantiated by treating the inequality of two states as a fact while using the state-equivalence assumption between them as a belief. Having two abstracted states $\underline{s}_i^{'} = \underline{s}_i$, as well as two concrete input segments $(s_i, (x_{val}, dt)) \neq (s_i^{'}, (x_{val}^{'}, dt^{'}))$, then the abstract candidate input segment $(\underline{s}_i^{'}, (x_{val}^{'}, dt^{'}))$ and the abstract observed input segment $(\underline{s}_i, (x_{val}, dt))$ are equal (cf. Figure 5.5). That is

$$(\underline{s}_i, (x_{val}, dt)) = (\underline{s}_i^{'}, (x_{val}^{'}, dt^{'})).$$

Now, we can simply use the state-equivalence relation knowing that in fact we are using $s_i \neq s_i^{'}$ together with the state-equivalence assumption. Thus, we may say that the two concrete input segments are equivalent. I use the term equivalent in the sense discussed in Section 5.3.

In the example I have been discussing, state-equivalence is supported by abstraction of states. In general, we have:

$$(s_i, (x_{val}, dt)) \equiv (s_i^{'}, (x_{val}^{'}, dt^{'})), \qquad s_i \neq s_i^{'} \text{ or } dt \neq dt^{'} \text{ or } x_{val} \neq x_{val}^{'}.$$

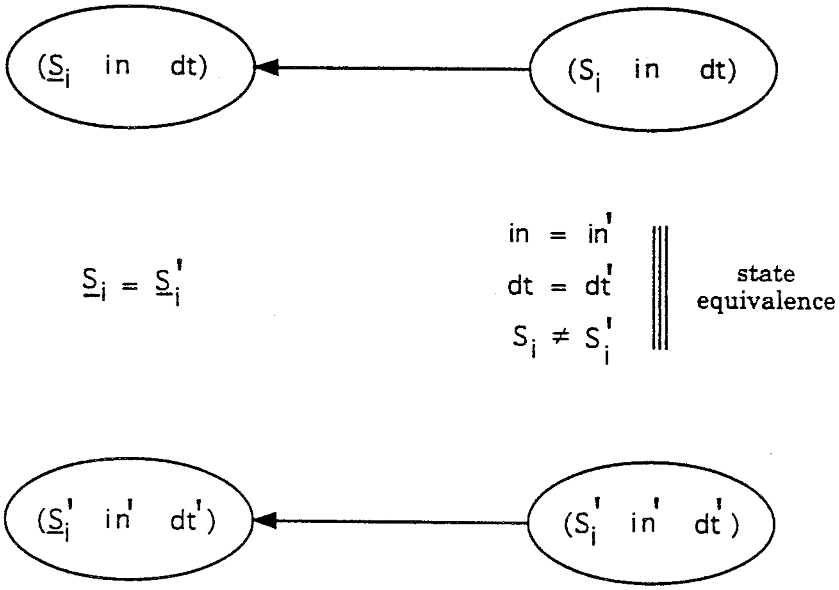


Figure 5.5, Abstraction of two input segments into their respective abstracted input segments

5.4.3 Predicting IO Segments

Although the concrete candidate input segment $(s_i', (x_{val}', dt')) \notin \widehat{F_G}$, we have seen its abstraction can be equal to the abstraction of an observed input segment. The equality between these two abstractions can be used to construct a new unobserved IO segment (cf. Figure 5.6) with the concrete candidate input segment and the output segment of the observed IO segment. That is, we can predict (construct)

$$((s_i', (x_{val}', dt')), \ (s_f, (y_{val}, dt))).$$

Here, I use the term *predict* to indicate that a predicted IO segment has no truth value assigned to it yet. Once the IO segment has been assigned a truth value, it is referred to as a *hypothesized* IO segment if its truth value is hypothesized, or an *asserted* IO segment if the segment has actually been observed. Therefore, an IO

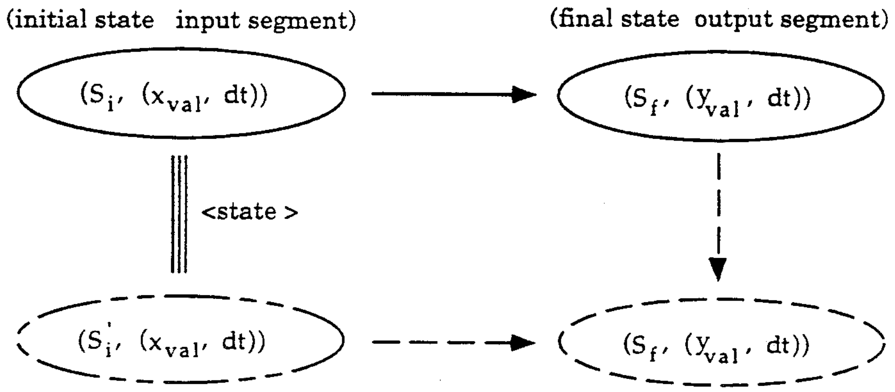


Figure 5.6, Constructing an unobserved IO segment

segment can be predicted, hypothesized, or asserted.

The supporting factors for a new yet unobserved IO segment are the observed IO segment $((s_i, (x_{val}, dt)),\ \ (s_f, (y_{val}, dt)))$, the state-equivalence assumption, and $\underline{s}_i = \underline{s}'_i$. In effect, we use the observed IO segment, the candidate input segment, and the state-equivalence relation to ignore (or abstract) the states of the input segments, in order to pair the candidate input segment with the output segment of the concrete observed input segment that is equivalent to it. A similar discussion can be given for constructing unobserved IO segments based on other equivalence relations.

Earlier, I supposed truth assignments for input segments in order to discuss under what conditions two unequal input segments can be considered equivalent, and consequently, to construct an unobserved IO segment. The question is whether the IO segments' truth assignment alone is sufficient or not. The answer primarily depends on what is communicated to the LTMS. There is no need to discuss the truth assignments of input segments as long as the hypothesized IO segments reflect their truth values. Each IO segment pair has a list representation for the $\widehat{G_F}$ and a logical representation for the LTMS.

We have already decided that the LTMS represents IO segments and not their parts. Thus, we only need to be concerned about the truth assignment of the IO segments. Each IO segment pair represents either a fact or a belief. Those IO segments that have been observed are assigned the truth value TRUE. In this way, each IO segment is known to the LTMS as a datum with its truth value TRUE. Furthermore, it is declared as a fact. I don't consider any negative facts, i.e., IO segments that can be assigned once and for all the truth value FALSE. This is in agreement with the data representation chosen in the iterative IOFO structure. It only contains the observed IO segments (positive facts).

What about those IO segments that are to be hypothesized? A hypothesized IO segment can have any of the truth values TRUE, FALSE, and UNKNOWN. Their truth values (beliefs) must, however, be justified. For instance, in the above example, if the two input segments satisfy $x_{val} = x'_{val}$ and $s_i \neq s'_i$, and if we use the state-equivalence relation to hypothesize $(s_f, (y_{val}, dt))$, we would declare the hypothesized IO segment as an assumption to the LTMS. Note that I am using the term "assumption" here differently from its use in the Truth Maintenance System. It is therefore better to call a hypothesized IO segment a *hypothesis*, as we already referred to the state-equivalence as an *assumption*. Every hypothesized IO segment must have an assigned truth value. The truth value of any hypothesized IO segment is TRUE at first, since the intention of hypothesizing an IO segment is also to believe in it!

It should be noted that, although the original truth value of a hypothesized IO segment is always TRUE, the semantics of this assignment are completely different from those of the truth value of an observed IO segment pair. A hypothesized IO segment's truth value is substantiated by a state-equivalence assumption, for example. That is, the original truth value of a hypothesized IO segment (TRUE) is subject to revision if there is knowledge to the contrary. Hence, while an asserted IO segment's truth value can only be TRUE, the truth value of a hypothesized IO segment can be TRUE, FALSE, or UNKNOWN. Furthermore, when the assigned truth value of a candidate input segment is revised, any previous (tentative) decisions derived based on it become subject to revision as well.

It would, of course, be possible to communicate more detailed information to the LTMS, such as the truth value of an individual input segment, or even the facts that $x_{val} = x'_{val}$ and $dt = dt'$ whereas $s_i \neq s'_i$. Such knowledge might be useful in more elaborate types of reasoning than what I am presently discussing. However in the present DIR implementation, each IO segment is communicated to the LTMS as a single entity representing either a fact or a belief having a single truth value assigned to it.

5.4.3.1 Consistency Axioms

Thus far, we have been predicting IO segments based on observed IO segments only. However, it is also possible to predict IO segments based on previously hypothesized IO segments. As I have indicated, the DIR is able to add to its database new IO segments incrementally. These new IO segments can either reflect new observations or be the product of previously made hypotheses.

A major role of the DIR is to assure the consistency among all of its IO segments. All observed and hypothesized IO segments must be consistent since the LTMS maintains a single database. What does the inference engine do? Essentially it has to compare a *candidate* (predicted or observed) IO segment with the *existing* (hypothesized or asserted) IO segments and determine whether the candidate IO segment should be added to the database, or not; and if so, whether it should be treated as a fact (asserted) or a hypothesis (hypothesized). To this end, the inference engine needs to interact with the LTMS, since the latter maintains all the previously asserted and hypothesized IO segments. Each candidate IO segment represents either an observation or a prediction. Likewise, each existing IO segment is either asserted or hypothesized (cf. Figure 5.7.)

In order to ensure that the databases of the inference engine and of the LTMS are never contaminated by inconsistent IO segments, every candidate IO segment has to be examined against all existing IO segments for consistency before it can be added to the databases. To check whether a candidate IO segment is consistent with a current entry in the databases, we need *Consistency Axioms.* The role of these axioms is to enforce both causality and uniqueness of all IO segments.

Recall that $\widehat{F_G}$ is a function. We can also perceive the collection of hypothesized IO segments as another set of functions. In the following section, I give a precise description of an overall structure containing all the observed and hypothesized IO segments while enforcing their consistency.

How do we ensure consistency between a candidate IO segment and an existing IO segment? On the one hand, observed IO trajectories are facts, and thus, cannot have their truth values revised. On the other hand, hypothesized IO segments are beliefs, and thus, their truth values can be revised under certain conditions. The inference engine has to determine whether the candidate IO segment is to be communicated to the LTMS as an assertion or as a hypothesis. Correspondingly, it also has to determine the type of the existing IO segment, which is either asserted or hypothesized. Once the inference engine knows the types of IO segments being compared, it needs to consider one of the cases shown in Figure 5.7.

CASE	*Candidate IO segment*	*Existing IO segment*
I	observed	assertion
II	observed	hypothesis
III	predicted	assertion
IV	predicted	hypothesis

Figure 5.7, Types of IO segments to be compared

A consistent candidate IO segment with respect to all existing IO segments can become:

- an assertion, or
- a hypothesis, or
- a redundant assertion or hypothesis,

and an inconsistent candidate IO segment is either:

- rejected, or
- undecided

with respect to all existing IO segments. An undecided candidate IO segment indicates that the present knowledge of the DIR is insufficient to determine whether one or both of the candidate and the existing IO segments should be rejected.

I have not yet discussed under what conditions two IO segments are considered consistent. These conditions lead to the consistency axioms. I discuss only the details of consistency criteria (axioms) for Case III where the existing IO segment is "observed" and the candidate IO segment is "predicted". The basic ideas are equally applicable to the other cases. Let us denote these IO segments as:

$$\begin{aligned} IO_{candidate} &= ((s_i^{'}, (x_{val}^{'}, dt^{'})),\ (s_f^{'}, (y_{val}^{'}, dt^{'}))) \\ IO_{existing} &= ((s_i, (x_{val}, dt)),\ (s_f, (y_{val}, dt))). \end{aligned}$$

We need to determine what conditions are necessary to declare two IO segments consistent. If only the initial states or the input events of two IO segments are equal, then they do not need to be examined for consistency — the two IO segments are consistent with respect to each other. That is, given two IO segments, if either of the following two conditions holds, they are consistent.

$$s_i^{'} \neq s_i, \quad \text{or}$$

$$x_{val}^{'} \neq x_{val}.$$

Consistency here simply means that the two compared IO segments represent different yet compatible situations that can coexist in the databases.

However, if both the initial states and input events of two IO segments are equal, then they must be examined for consistency. Two input segments with identical initial states and input events can differ, depending on their durations. In particular, three possibilities can occur given the durations of the candidate IO segment $IO_{candidate}$ and the existing IO segment $IO_{existing}$. We have:

$$\begin{aligned} &(a) \qquad \ell(IO_{candidate}) = \ell(IO_{existing}) \\ &(b) \qquad \ell(IO_{candidate}) < \ell(IO_{existing}) \\ &(c) \qquad \ell(IO_{candidate}) > \ell(IO_{existing}) \end{aligned}$$

In all three cases, $s_i^{'} = s_i$ and $x_{val}^{'} = x_{val}$. In case (a), if both output events are *nil*, then the two IO segments are identical, and the predicted IO segment is *redundant*. This, of course, would never happen if the IO segment were predicted — there would not have been any reason for predicting it.

However, if their output events do not match, i.e.,

$$dt^{'} = dt$$
$$y_{val}^{'} \neq y_{val}$$

the two IO segments are inconsistent. In this case, the predicted candidate IO segment must be *rejected*. This means it should not be added to the databases. Of course, it could also be asserted, provided its truth value is FALSE. (Note that even if we decide to include it in the databases, it must not be made a hypothesis since the durations of both IO segments are equal.)

In case (b), the IO segments are consistent when their output events are *nil*. In other words, when axiom (5.1) is satisfied, the two IO segments are consistent. That is,

$$dt^{'} < dt$$
$$y_{val}^{'} = y_{val} = \phi \tag{5.1}$$

If the candidate IO segment's output event is *nil* and the existing IO segment's output event is not *nil*, the predicted IO segment is also consistent. This leads to axiom (5.2)[4]

$$dt^{'} < dt$$
$$y_{val}^{'} = \phi \qquad \text{and} \qquad y_{val} \neq \phi \tag{5.2}$$

When either of the above two axioms is satisfied, the predicted IO segment is *hypothesized* with truth value TRUE.

[4] Note that even though axioms (5.1) and (5.2) can be combined to $dt^{'} < dt$ and $y_{val}^{'} = \phi$, it is better not to hide the distinction between them.

The remaining two axioms for case (b), (5.3) & (5.4), show under what conditions two IO segments are inconsistent. When either of these two axioms is satisfied, DIR rejects the candidate IO segment.

$$\begin{gathered} dt^{'} < dt \\ y^{'}_{val} \neq \phi \quad \text{and} \quad y_{val} = \phi; \end{gathered} \tag{5.3}$$

$$\begin{gathered} dt^{'} < dt \\ y^{'}_{val} \neq \phi \quad \text{and} \quad y_{val} \neq \phi. \end{gathered} \tag{5.4}$$

In case (c), the IO segments are also consistent when the output events are *nil.* The axiom for this situation is similar to (5.1); we have:

$$\begin{gathered} dt^{'} > dt \\ y^{'}_{val} = y_{val} = \phi. \end{gathered} \tag{5.5}$$

If the candidate IO segment's output event is not *nil* and the existing IO segment's output event is *nil*, they are consistent as well. This situation is the reverse of axiom (5.2).

$$\begin{gathered} dt^{'} > dt \\ y^{'}_{val} \neq \phi \quad \text{and} \quad y_{val} = \phi. \end{gathered} \tag{5.6}$$

Once again, the predicted IO segment will become a hypothesis. Any of the remaining two axioms, (5.7) & (5.8) — similar to (5.3) & (5.4) — when satisfied, indicates to the DIR that the predicted IO segment and the asserted IO segment are inconsistent. As usual, if the predicted IO segment is inconsistent with the observed IO segment, it is rejected.

$$\begin{gathered} dt^{'} > dt \\ y^{'}_{val} = \phi \quad \text{and} \quad y_{val} \neq \phi; \end{gathered} \tag{5.7}$$

$$\begin{gathered} dt^{'} > dt \\ y^{'}_{val} \neq \phi \quad \text{and} \quad y_{val} \neq \phi. \end{gathered} \tag{5.8}$$

Hence for Case III, we say that two IO segments are *compatible* when one of the axioms (5.1), (5.2), (5.5), or (5.6) is satisfied.

In determining the consistency between two IO segments, I did not examine the equality (or inequality) of the final states of the predicted and observed IO segments. What are the implications of not doing so? Of all the situations I considered, we need to be concerned with those four of the above axioms that result in declaring a predicted IO segment as consistent. I discuss only one of them.

Given (5.1) with inclusion of the final states of the IO segments, we are confronted with either of the following two situations:

$$\begin{gathered} dt' < dt \\ y'_{val} = y_{val} = \phi \\ s'_{f} = s_{f} \end{gathered} \qquad (5.1.a)$$

$$\begin{gathered} dt' < dt \\ y'_{val} = y_{val} = \phi \\ s'_{f} \neq s_{f} \end{gathered} \qquad (5.1.b)$$

We may wish to declare a predicted IO segment as consistent only when axiom (5.1.a) is satisfied. In the case of axiom (5.1.b) being satisfied, we may decide to declare the predicted IO segment as inconsistent. Such a decision would be supported by taking the view that, while both outputs remain *nil*, their corresponding final states should remain unchanged. What this requires is that an observed output of a system must be a direct reflection of its past — different outputs correspond to different time histories. If we impose this restriction, we are claiming that only certain types of outputs are observed, namely those that are strongly related to states. Imposing such a demand on output trajectories is rather strong. I have chosen to take the opposite view — (5.1) should not be replaced by (5.1.*a*) and (5.1.*b*). Of course, it would be a simple matter to replace (5.1) by the more restrictive axioms (5.1.*a*) and (5.1.*b*). Similar explanations can be given for axioms (5.2), (5.5), and (5.6).

Now, we say that two IO segments are consistent when (1) they are equal, (2) either the initial states or the input events of their complete input segments are unequal, or (3) they are compatible. Consistency checking based on (1) and (2) are the same for all four cases (cf. Figure 5.7). Consistency checking among these four cases differs only in (3), i.e., in the interpretation of compatibility between IO segments. For Case III, we showed explicitly under what conditions two IO segments are said to be compatible with each other.

The consistency of a candidate IO segment against an existing IO segment depends on the types of the candidate and the existing IO segments. My intention at this point is to describe what might happen to the candidate/existing IO segments for each of the four cases. Due to previous exposition of what consistency between IO segments entails in general, I am now able to do so without going into any details of what conditions need to be satisfied precisely in each of the four cases in order to declare two IO segments consistent with each other.

Let us first assume that the candidate IO segment is observed. Then, there are two possibilities (Case I & Case II) depending on whether an existing IO segment is observed (an assertion) or hypothesized (a hypothesis).

In Case I, we have a contradiction between observed IO segments if they are inconsistent. When this happens, additional knowledge is required to decide how to resolve the inconsistency. An IO segment that is *undecided* (i.e., cannot be determined given the available IO segments) is denoted by a "$\star$". If two observed

IO segments are found to be inconsistent with each other, then both will be marked with a "$\star$". Note that the entire database needs to be searched to check whether other previously observed IO segments need also to be marked with a "$\star$".

Observed (asserted) IO segments are not dependent on any assumptions. If no inconsistency is discovered between the candidate IO segment and any of the previously observed IO segments, then the candidate IO segment is either redundant (consistent with one or several previously observed IO segments) or has been observed for the very first time. In the former case, the redundant IO segment is *ignored.* In the latter case, the newly observed IO segment is *asserted.* Note that an observed IO segment is always consistent with itself; that is, the newly observed IO segment can be added to the databases as asserted if no compatible IO segment has been found.

Inconsistencies between a newly observed IO segment and the body of previously asserted IO segments takes precedence over all other considerations. However, since the database is consistent at all times, it is possible to search for equality and inconsistencies simultaneously. If a previous entry is found that is equal to the candidate, then the candidate can be discarded as a duplicate, and the search can stop at once. If, on the other hand, an inconsistency has been discovered, it is important to continue the search through the database for other potential inconsistencies. Only if neither an equality nor an inconsistency has been found in the entire database, should the database be searched in a second pass for compatible entries.

The Case I situation takes precedence over Case II. Thus, only after the entire database has been searched twice for assertions, and the observed candidate IO segment has been asserted (i.e., has been found to be consistent and not redundant with the body of previous experiences) should the database be searched for hypotheses.

When the database is searched for hypotheses, the candidate IO segment has already been communicated to the LTMS as a fact with truth value TRUE irrespective of whether it is consistent with any or all of the previously stored hypotheses.

If a hypothesis is found that is equal to the newly asserted IO segment, then the existing IO segment is converted from a belief to a fact, and since it is redundant with the newly asserted IO segment, one of them can be deleted from the databases.

If a previously TRUE hypothesis is found to be inconsistent with the newly asserted IO segment, its truth value changes from TRUE to FALSE, and since FALSE hypotheses are not overly useful, I have chosen to eliminate them from the databases for the benefit of compactness of the LTMS.

Previously TRUE hypotheses that are found to be compatible with the new assertion will remain unchanged, i.e., will remain TRUE hypotheses.

Let us now look at the case where the candidate IO segment represents a prediction. Since assertions are always stronger than hypotheses, the Cases III and IV are a little simpler to handle than the Cases I and II. None of the decisions made relative to a candidate prediction will ever make the database inconsistent. Consequently, it suffices to handle the Cases III and IV in a single pass through the database, and I can thus focus my discussion on a comparison of individual pairs.

Note that, in Cases III and IV, the inference engine is not supposed to hypothesize IO segments that have either been previously observed or predicted; there would be no reason to do so! Consequently, if the candidate is found to be equal to an existing hypothesized IO segment, the already existing hypothesized IO segment is *confirmed*; and if it is equal to an existing observation, then it is discarded (left unchanged). Obviously, in either case, the search can stop immediately.

Let's discuss Case III now. If an existing IO segment is an assertion and is inconsistent with the predicted candidate IO segment, the inference engine has two choices: (1) communicate the predicted IO segment to the LTMS as a fact with truth value FALSE, or (2) simply reject it. I have chosen to throw away inconsistent predictions since there are a great many of them. If an existing IO segment is compatible (yet not equal) with the predicted candidate IO segment, then the predicted IO segment is communicated to the LTMS as a hypothesis (belief) with truth value TRUE.

In Case IV, two choices are possible for the inference engine as well. (1) If they are compatible, then the predicted candidate IO segment is communicated to the LTMS as a hypothesis with a truth value of TRUE, and the existing hypothesized IO segment remains unchanged. (2) If they are inconsistent, then further knowledge and reasoning are once again necessary. In this situation, the existing and the predicted IO segment cannot both be true, but if one of them is true, we don't know which one it is. Finally, it is also possible that both are false, and this is the simplest choice the inference engine can make. In our implementation of the DIR (discussed in Chapter 6), I have chosen to let the user resolve the contradiction.

Figure 5.8 summarizes what effects the consistency axioms may have on the IO segments in all four cases shown in Figure 5.7. In the left column of Figure 5.8, a *Candidate IO Segment* may be:

$$\{\text{asserted, hypothesized, rejected, ignored,}\star\}.$$

Likewise, in the right column of Figure 5.8, an *Existing IO Segment* may be:

$$\{\text{confirmed, rejected, unchanged,}\star\}.$$

Note that the elements in each of the above sets are mutually exclusive. For instance, when an IO segment cannot be asserted, hypothesized, confirmed, ignored, or left unchanged due to an inconsistency situation, it is denoted as $\star$.

It should now be apparent why the consistency axioms are paramount to the proper functioning of the truth maintenance system. The DIR uses the consistency axioms in order to update the databases with newly observed or predicted IO segments while maintaining consistency among all the prior IO segments, both asserted and hypothesized. The consistency axioms serve two purposes. First, they ensure consistent databases. Second, their use prevents the databases from getting too large by ignoring candidate IO segments that are redundant with previously stored IO segments.

So far, I have discussed how the DIR can predict IO segments. Does the DIR always predict a unique IO segment? No. Due to the abstraction, multiple IO segments may be predicted for a candidate input segment. For instance, if the state-equivalence assumption is used, it might be that there are multiple observed IO segments that match the duration and input value of the candidate input segment, thus resulting in multiple predicted IO segments.

How do we handle this situation? Non-monotonic reasoning basically offers us three possibilities. We could decide to make multiple predictions and pursue all of them further, as e.g. QSim would [Kui86]; or we could use a measure of likelihood to determine which prediction looks most promising, as e.g. SAPS would [Cel91]; or finally, we might make further assumptions until only one prediction is left. In Section 5.4.5, I shall discuss how DIR reacts in this situation. It shall be shown that the user has a choice between options one and three, and how precisely option three is implemented.

Candidate IO Segment	Existing IO Segment
Observed {asserted, ignored, $\star$}	**Assertion** {unchanged, $\star$}
Observed {asserted}	**Hypothesis** {confirmed, rejected, unchanged}
Predicted {hypothesized, rejected}	**Assertion** {unchanged}
Predicted {hypothesized, $\star$}	**Hypothesis** {unchanged, $\star$}

Figure 5.8, How candidate and existing IO segments may fare w.r.t. to the consistency axioms.

5.4.4 Partitioning Input Trajectories

I still need to describe how an input trajectory containing multiple events is partitioned into input segments. This is necessary since we only know how to reason with IO segments, not with trajectories.

Note that for now, I shall concentrate on the input trajectories alone, since we wish to study the process of *prediction*, i.e., the situation where no output trajectory is available. The case of ***observations*** of input/output behavior of a system for the purpose of augmenting the databases is different. In that case, we would need to segment the input and output trajectories simultaneously, since the duration of an input segment would then be influenced by events occurring in the output trajectory as well.

Let us call a *candidate* input segment an input segment of either type 1 or 2 as defined in Section 5.2. Given an input trajectory, the input event of the first candidate input segment is the first input event of the input trajectory. Its duration is measured as the duration between the first and second input events of the input trajectory. Likewise, the input event of the second candidate input segment is the second input event of the input trajectory. Obviously, for any input segment ω, $\ell(\omega) > 0$. I shall suppose that, for each input segment, ω has a duration of either *unit length* or a multiple thereof, where unit length is determined based on assumption set-I (cf. Sections 3.2 and 5.2).

I use the term *partition* to refer to the process of dividing an input trajectory into candidate input segments. Of course, an input trajectory as well as any Complete input segment has associated with it an initial state. However for now, our discussion is based on input segments such as ω (see Section 5.2), and not on Complete input segments. An input trajectory without initial state is represented as:

$$in\_traj\_i : ((x_{val,1},\ dt_1), (x_{val,2},\ dt_2), \ldots, (x_{val,n},\ dt_n)).$$

I use an example instead of the generic input trajectory in_traj_i to make the discussion easier. Let

$$in\_traj\_1 : ((c,\ 1), (a,\ 5), (b,\ 2)),$$

be an exemplary trajectory, where $x_{val,1} = c$, $dt_1 = 1$, $x_{val,2} = a$, $dt_2 = 5$, and so on. This input trajectory can be partitioned into the following candidate input segments

$$in\_seg\_1 : (c,\ 1)$$
$$in\_seg\_2 : (a,\ 5)$$
$$in\_seg\_3 : (b,\ 2).$$

Let us look at one such candidate input segment. We have no reason to further partition the candidate input segment if there exists an IO segment (either asserted or hypothesized), the input segment of which matches the candidate input segment. However, if neither an asserted nor a hypothesized matching IO segment can be found in the database, then the candidate input segment (of length greater than unit length) needs to be further partitioned.

The partitioning of a candidate input segment into two *smaller* input segments allows the composition of the original candidate input segment from these smaller input segments. The first smaller input segment is referred to as *left input segment* and the second input segment as *right input segment* (cf. Section 2.1.4). That is, given $(x_{val},\ dt)$, its left and right input segments become $(x_{val,left},\ dt_{left})$ and $(x_{val,right},\ dt_{right})$ where $x_{val,left} = x_{val}$, $x_{val,right} = nil$, and $dt = dt_{left} + dt_{right}$. When the input segment cannot be partitioned any further, it becomes the left input segment. An input segment is called smaller w.r.t. another input segment when its duration is less than that of the other input segment. The rationale behind partitioning a candidate input segment into left and right input segments is that either (1) there exist (asserted or hypothesized) IO segments with their input segments matching the left input segment, or (2) compatible IO segments can be predicted based on the left input segment. Note that I am also using the term partition to refer to subdividing a candidate input segment with duration greater than unit length into shorter input segments.

When the duration of a candidate input segment is greater than unit length, it may be partitioned in several ways depending on the available IO segments. Given a candidate input segment, we partition it based on three choices — `longest`, `exact`, and `all`, indicating the desired duration for any left input segment.

Imagine we have the candidate input segment $in\_seg\_2 : (a,\ 5)$ with $x_{val} = a$ and $dt = 5$. I use this candidate input segment in discussing two of the above three choices. When the choice is `longest`, the candidate input segment's duration is reduced by a minimum number of unit lengths such that it matches either an asserted or a hypothesized IO segment (cf. Section 5.4.3.).

For instance, given $in\_seg\_2 : (a,\ 5)$ and provided that there are no matching IO segments with a duration of 5 unit lengths, the candidate input segment can be partitioned into left and right input segments where $in\_seg\_2_{left} : (a,\ 4)$ and $in\_seg\_2_{right} : (nil,\ 1)$, respectively. If, for $(a,\ 4)$, either a matching IO segment exists with its input segment equal to $(a,\ 4)$, or an IO segment can be hypothesized based on a previous assertion or hypothesis of shorter duration, then $(a,\ 4)$ becomes the "longest" input segment. If neither is possible, then $(a,\ 5)$ is partitioned such that its *next* longest left input segment and its corresponding right input segment are $(in\_seg\_2_{left} : (a,\ 3))$ and $(in\_seg\_2_{right} : (nil,\ 2))$, respectively. Again it might be that $(a,\ 3)$ is the longest input segment. If not, the same process is carried out until the "next longest" input segment leads to a matching IO segment, or until the duration of the left input segment is unit length.

Of course, it can happen that the "longest" IO segment actually has its duration equal to one of the available IO segments. That is, when the choice is `longest`, the goal is to attempt to find an IO segment with the maximum duration possible. The DIR will try to find an IO segment of the same length as the candidate input segment; if none can be found, then an IO segment of shorter duration will be acceptable also. Once a left input segment leads to a consistently predicted IO segment, its right input segment becomes the new candidate input segment. The iteration process stops at the latest when the last left input segment is of unit length duration and when no IO segment can be predicted for a left input segment with its duration equal to unit length.

The partitioning of a candidate input segment is not necessarily unique. It is possible that a left input segment with shorter duration (in comparison to the one with the longest duration) may also lead to a consistent IO segment. In particular, we may want a candidate input segment to be partitioned such that its left input segment has a length *equal* to one of the available IO segments. Of course, there may exist several (or many) IO segments with durations of unit length and greater. I use the term *exact* to denote our preference to choose the left input segment such that its duration is *equal* to the duration of one of the available IO segments. Among those, we shall select the one with largest duration. That is, the choice `exact` not only requires that the duration of the left input segment of a partitioned candidate input segment be equal to an available IO segment, but among those IO segments that satisfy this condition, the one chosen will be one of maximum duration. In other words, the choice `exact` implies that no hypothesized IO segment for a left input segment can have its duration greater than any of the available IO segments.

For example, suppose the above *in_seg_2* is partitioned into $in\_seg\_2_{left}$: (a, 3) and $in\_seg\_2_{right}$: (nil, 2), and there exists an IO segment such that its input segment forms a perfect match or an IO segment can at least be consistently hypothesized for (a, 3) based on an IO segment with a duration of 3 unit lengths. In either case, the duration of the left input segments is equal to that of the exiting IO segment, and the `exact` specification is satisfied. If no such IO segment can be found, the candidate input segment will have to be partitioned differently, namely with its left input segment being of shorter duration than 3 unit lengths.

Finally, an input segment with length greater than unit length can be partitioned in `all` possible forms. In this case, the DIR may provide multiple predictions, leading to the branch-out problem that is so well known from qualitative physics [Bob84]. I won't provide the details of this selection since it represents a straightforward extension of the previous two choices.

I emphasize that it is not always possible for every candidate input segment to be partitioned such that the "longest" ("exact") left input segment can lead to a consistent IO segment. If this happens, the "next" longest or exact input segment for a candidate input segment is computed. Also, partitioning of a candidate input segment is recursive. That is, it is possible that once the longest (exact) left input segment is determined, the right input segment's duration is greater than unit length, which in turn may be partitioned accordingly.

Finally, I caution the reader that I have not been explicit about the role of initial states in partitioning of an input trajectory or a candidate input segment. Whenever a candidate input segment or a left input segment leads to a consistent IO segment, its final state becomes the initial state of the subsequent left input segment.

All three choices in partitioning an input trajectory are different variations for `abs-length`. In particular, depending on the available observed IO segments and the assumption set `asn-set`, only a subset of all possible partitioned input segments eventually can result in predicted IO segments.

Earlier in this chapter, I presented different types of abstractions on a segment's length, input event, and state. Now with these three choices on the type of `abs-length`, I can succinctly specify assumption set-II. It has two elements: `elm-1` $\in$ {`longest`, `exact`, `all`} and `elm-2` $\in$ {`abs-input`, `abs-state`} (cf. Section 5.3 for the elements of `elm-2`). Therefore, each assumption specifies the type of `abs-length` from `elm-1`, and whether input event or initial state should be abstracted if necessary. I use the qualifier necessary in the previous sentence to indicate that input (or state) abstraction should be used only if abstraction on length alone does not lead to any hypothesized IO segment. As an example, `(longest, abs-input)` would be a legitimate choice for the assumption set-II.

5.4.5 Predicting Output Trajectories

I still need to describe how the DIR generates one or more output trajectories for an input trajectory. Here I shall only provide a top-level description of how this process works, while leaving the details for the following chapter.

Given an input trajectory, an initial state, a choice of assumption set-II, and some available (asserted and hypothesized) IO segments, the task of the DIR is to predict at least one output trajectory for the given input trajectory. As mentioned previously (Section 5.4.3.1), it is possible that multiple IO segments can be hypothesized for a single input segment. When this is the case and if `elm-1: abs-all`, then all of these possibilities are explored to construct multiple output trajectories. This corresponds to the way QSim [Kui86] predicts the future. However, when `elm-1` ∈ {`longest`, `exact`}, then only one among the multiple possibilities will be pursued at any one time. This resembles the way in which SAPS [Cel91] predicts the future. However, rather than relying on an assessment of the likelihood of a particular prediction (the statistical approach), DIR relies on two assumption sets in selecting the prediction to be pursued (the artificial intelligence approach).

Note that assumption sets are never exhaustive. It is possible (and it may actually happen quite often), that the chosen assumptions do not lead to a single prediction. For example, if the chosen assumption set-II is `(longest, abs-input)`, it may well happen that, even without input abstraction, two different longest input segments of equal length result. In this situation, our implementation of the DIR will select one at random. SAPS has the same problem. If the two most likely predictions have the same value of likelihood, SAPS will pick randomly either of the two.

The inference engine begins by finding the candidate input segment and forms a Complete candidate input segment using the given initial state. Then, it tries to find a match for the Complete candidate input segment in the database of previously asserted IO segments. If it succeeds, it extracts the final state of the newly found IO segment and uses it as the initial state for a new input trajectory with the previous candidate input segment chopped off from the previous input trajectory. Now, another candidate input segment is computed and the same process is repeated if applicable.

As I have discussed, it is also possible that there may not exist an observed IO segment that matches the Complete candidate input segment. In this case, the DIR examines the hypothesized IO segments for a match. If it succeeds, it follows the steps described in the above paragraph with the exception that the match is with a hypothesized IO segment instead of an asserted IO segment. Evidently, it makes sense to give preference to asserted IO segments over hypothesized ones.

If the DIR cannot find a match for the candidate input segment from either the asserted or the hypothesized IO segments, it then attempts to hypothesize an IO segment on the basis of compatible, yet not equal, asserted or hypothesized IO segments, and communicates this hypothesis with its assigned truth value to the LTMS in accordance with the assumption set used. As indicated earlier, when length alone suffices to hypothesize an IO segment, then the other part of the assumption set-II (i.e., abstraction on `abs-input` or `abs-state`) is not used. Once again, given the hypothesized IO segment, a new input trajectory is constructed. The initial state for the new input trajectory is the final state of the hypothesized IO segment. This process iterates using the new input trajectory and the new initial state until the concatenation of all selected input segments is the same as the original input trajectory or until no IO segment can be hypothesized for an intermediate candidate or left input segment. In the former situation, at least one but possibly several output trajectories can be predicted. In the latter situation, the prediction process comes to a halt when no IO segment can be hypothesized for a given input segment, thus providing only partially predicted output trajectories. SAPS has the same problem. If it runs into a state that it has never seen before, the forecasting process comes to a grinding halt. The problem can be cured by adding more evidence (IO segments) to the experience databases. However, our approach is somewhat more flexible in this respect than SAPS, because the problem can also be solved if no additional data are available, simply by allowing stronger abstraction types to be used by the DIR. This technique can be employed to alleviate the lack of sufficient data.

The DIR, of course, can receive additional observed IO segments. In that case, it should be able not only to update its databases, but also to retract any hypothesized IO segment that may contradict the newly received observed data. That is, when any newly observed IO segment α contradicts any previously hypothesized IO segment $\alpha^{'}$, the DIR concludes that $\alpha \wedge \alpha^{'} \vdash \perp$. The DIR resolves this contradiction by assigning the truth value FALSE to $\alpha^{'}$.

A final note is for the situation where contradictions are due to inconsistencies between a previous assertion and a newly observed IO segment, or between a previously hypothesized IO segment and a newly predicted one. We already discussed these situations in Section 5.4.3.1. The $\{\star\}$ instructs the DIR to be cautious when using such evidence in concluding anything. It will do so only as a last resort. The $\{\star\}$ represents a mild form of inconsistency that is tolerated to migrate into the LTMS, because it cannot be avoided.

This concludes the basic description of the DIR. Of course, we don't know yet how powerful the DIR really is. It could be that the reasoning mechanisms used are much too rigid, and therefore, any prediction will invariably come to a halt. It could also be that the reasoner behaves like the oracle of Delphi, predicting in every situation that "tomorrow the weather will change or remain the same as today." Chapter 6 shall address this question at least in an experimental fashion by showing the actual implementation of our DIR and by presenting realistic examples of its use.

Chapter 6 LDIR: An Implementation of DIR

A theory aimed at providing a basis for a computational model can become dormant until an *implementation* of it demonstrates at least some of its claims. In the introduction, I discussed the an ongoing debate between the virtues of approaches followed by the experimentalists and the formalists. To settle the dispute, the formalists could support their "theories" with counterpart "implementations"; alternatively, the experimentalists could do the converse — i.e., devise "proofs" for their "experiments." [Kuh62, Hay73, McD76, RH84] point, for example, to the lack of coherency between the ideas expressed by the formalists and the results presented by the experimentalists when the two are developed separately.

At the outset of this writing, I indicated that one of my goals is to implement a subset of a discrete-event inductive reasoner. Having an implementation enables us to examine what aspects of the theory is actually working and useful, how the theory works, and why it works. Furthermore, an implementation provides a playground for exploring ideas that have not been developed formally.

Consequently, this dissertation does not fall either in the camp of the formalists or in that of the experimentalists in any strict sense. The dissertation started out in a rather formalistic way; in due course, it shifted toward the experimental side, while making full use of the formerly introduced theories in the procedures described to perform the experiments. This approach did not simply happen. The course of action was chosen deliberately to narrow the gap between the two camps.

I begin by describing an example to illustrate the underlying mechanisms of the implementation of the DIR. I then describe an implementation of the DIR, called *Logic-based Discrete-event Inductive Reasoner (LDIR)*. I use fragments of two observed I/O trajectories (*io-traj-1* and *io-traj-2*) to construct a database, and an input trajectory (*in-traj-3*), for which output trajectories are to be predicted. Key features of LDIR's implementation shall be discussed by means of this simple yet non-trivial example. Several predicted output trajectories for the input trajectory *in-traj-3* will be presented and analyzed in detail. They shall serve to evaluate LDIR's performance.

6.1 Example: A Shipyard

Imagine there exists a shipyard[1] that contains two Repair Stations: RS-1 and RS-2. The shipyard receives different types of vessels in need of repair. To simplify the example, assume only one of the two repair stations is expected to be in operation during any given period. Let us assume that RS-1 adheres to a First-In-First-Out (FIFO) discipline, whereas RS-2 observes a Priority Ranking (PR) discipline whereby vessels of type "B" are given priority over boats of type "C" and ships of type "A" have the highest priority of all. Since only one repair station is in use at any point in time, an observer sitting on the shore may observe a FIFO discipline for two weeks, and a PR discipline for the next two.

Vessels in need of repair enter the shipyard at stochastically chosen time points, wait for their turn, and depart the shipyard after being repaired. Each vessel type is assigned an I.D. number indicating how much time is required for it to be fixed. In this simple example, there are three types of vessels — "A," "B," and "C" — that require one unit, two units, and three units of time, respectively, for repair. Only one vessel is being repaired at any one time.

Next, suppose an observer records the arrival and departure of vessels to a shipyard. The arrival of vessels corresponds to input events, and the departure of vessels corresponds to output events. An observer may record the number of vessels that enter the repair stations. The number of vessels in the shipyard is

[1] Originally, I used a bank with two tellers as an example. During a conversation with Alex Meystel, he laughingly said we are not businessmen, are we? He then suggested I call the bank a shipyard instead. Of course, the dynamics of the two systems are essentially the same (at least at the level of abstraction dealt with here), even though aesthetically they are not!

incremented (decremented) by one whenever a vessel enters it (departs from it). Instead of specifying the number of vessels in the repair station, the observer may keep an ordered record of the vessels entering the repair station, while removing from the list any vessel departing from it.

However, in order to completely specify the state of a repair station, it is necessary not only to have an ordered record of the vessels entering the repair station, but also to specify how much time is necessary for each vessel until it is able to depart again, given that some vessels are already waiting for repair. For example, even though a vessel of type "A" requires only one unit of time for repair, it might require two units of time if it has to wait.

LDIR has been implemented in Common Lisp [Ste90] — now the de facto standard of Lisp programming languages. The list notation common to all Lisp languages was used to represent IO segments, IO trajectories, input trajectories, etc. Common Lisp, by default, generates all its output in upper case. Accordingly, I use an all-upper-case representation to denote LDIR output, and lower-case characters to denote everything else, including LDIR input and segments of Lisp pseudo-code simplified to increase the readability of the information presented. In particular, Lisp procedures are presented in a mathematical form. For example, an actual Lisp procedure such as

```
(defun ensure-consistency-asn (cand &optional asn-set &aux nodie)

              ...)
```

is shown in the following mathematically equivalent form:

```
ensure-consistency-asn(cand, asn-set)
```

In the above Lisp procedure, `ensure-consistency-asn` is the name of the procedure; `cand` is the required argument; `&optional` indicates the beginning of the list of optional arguments, here consisting of `asn-set` only, which, if omitted, is replaced by a default value; `&aux` indicates the beginning of the list of auxiliary variables, in the above example consisting of a single variable called `nodie`; and $\cdots$ represents the place where the actual computations of the procedure would be encoded. Since this chapter provides only a high-level presentation of the concepts, mechanisms, and procedural algorithms implemented in LDIR, the details of the Lisp code realizing these procedural algorithms are omitted in the interest of emphasizing more important conceptual issues, and a more readable mathematical pseudo-code representation is used in place of the actual Lisp code. In representing a pseudo-code, I don't distinguish between required and optional arguments, and I omit auxiliary variables altogether.

Within the framework of DEVS, the state of each repair station can be completely described by:

$$(\cdots,\ (id_j, t_j),\ \cdots)$$

where id_j denotes the identity of the vessel "J," and t_j denotes the remaining time before vessel "J" will depart from the repair station. Some abstractions of the above representation are:

$$\begin{gathered}(num\_of\_vessels,\ (\cdots,\ id_j, \cdots))\\(\cdots,\ id_j, \cdots)\\(num\_of\_vessels)\end{gathered}$$

where $num\_of\_vessels$ is the number of vessels that have entered the repair station and have not left yet. I chose the input/output trajectories for the shipyard to have as their states $(num\_of\_vessels,\ (\cdots,\ id_j, \cdots))$. Evidently, it is possible to compute $num\_of\_vessels$ from $(\cdots,\ id_j, \cdots)$. For convenience, I decided to make this information explicit. Each IO trajectory has as its input events the identities of arriving vessels. Likewise, the identities of leaving vessels are the output events of an IO trajectory.

An observed IO trajectory for the shipyard, represented in a pseudo Lisp notation, might take the form:

```
io-traj-1:  ((si-0 ()) (in a 1) (out a 1))
            ((si-0 ()) (in nil 1) (out nil 1))
            ((si-0 ()) (in b 2) (out b 2))
            ((si-0 ()) (in nil 1) (out nil 1))
            ((si-0 ()) (in c 3) (out c 3))
            ((si-0 ()) (in nil 1) (out nil 1)).
```

This particular IO trajectory is shown as the top IO trajectory in Figure 6.1. In the above representation, both `si-0` and `()` indicate that no vessel is at the repair station at the time of the input event. The pseudo-code `(in a 1)` indicates that a vessel of type "A" arrives at the current time. Since no vessel is in the shipyard, the total processing time for vessel "A" is equal to the servicing time of the vessel, namely one time unit. Since no other vessel is scheduled to arrive in between, this is also the duration of the first segment. The code `(out a 1)` shows the same vessel leaving at the end of the segment, i.e., one time unit later. It is not necessary to explicitly mention the final state associated with the output segment, since, because of continuity, the final state of one segment must coincide with the initial state of the next. Thus, the information is redundant and was suppressed in the above pseudo-code.

Similarly, had we encountered an initial state marked as `(si-3, (a b a))`, this would have indicated that, at the time of the new input event, there were three vessels currently already in the shipyard, two of type "A" and one of type "B."

The arrival or departure of no vessel at either the beginning or the end of a segment is indicated as nil, which was denoted as ϕ in earlier chapters[2]. For instance, (in nil 1) specifies that no new vessel arrived at the beginning of the segment.

Consequently, the first record of *io-traj-1* says that there were no vessels in the repair station initially, and that a vessel with identity a arrived at that time, which departed again one time unit later. The second record, ((si-0 ()) (in nil 1) (out nil 1)), says that no vessel arrived or departed during the second time unit and that no vessels were in the repair station during this time period, etc. Note that this particular IO trajectory satisfies both FIFO and PR displines.

Comparing the pseudo-code with the top graph of Figure 6.1, it should be very easy to interpret the pseudo-code correctly. An input trajectory such as *io-traj-1* is simply a list of records, each containing an initial state, an input segment, and an output segment, in the order given. The final state associated with each record is the initial state of the record following it, except for the last one. This is due to the underlying causality of the system. For this reason, I chose to end the trajectory description with a "nil" record, so that the final state of the trajectory would also be properly recorded.

Of course, the input and output segments of each record of a trajectory have the same initial states and the same final states. However, I chose to always associate the initial state of both segments with the input segment to form the Complete input segment. Likewise, the final state of both segments is always associated with the output segment constituting the Complete output segment.

Obviously, this same information could be represented in alternate forms as well. For instance, it would have been possible to record an IO segment using a syntax such as (((si-0 ()) (in b 1)) ((si-1 (b)) (out nil 1))) where ((si-0 ()) (in b 1)) corresponds to the Complete input segment, $(s_i, (x_{val}, dt))$,

[2] In Lisp, an empty list ( ) and the symbol nil are both returned as NIL. Nevertheless, I chose to represent in the pseudo-code an empty list as ( ) to distinguish it from the "nil" event.

with s_i = (si-0 ()), x_{val} = b, and $dt = 1$ (cf. Section 5-2), and where ((si-1 (b)) (out nil 1)) corresponds to the Complete output segment, $(s_f, (y_{val}, dt))$, with s_f = (si-1 (b)), y_{val} = nil, and $dt = 1$. Note that I used the identifiers in and out to differentiate between input and output events.

The second IO trajectory, which satisfies a FIFO discipline only, is shown in the bottom graph of Figure 6.1. Its pseudo-code representation is:

```
io-traj-2:  ((si-0 ()) (in b 1) (out nil 1))
            ((si-1 (b)) (in a 1) (out b 1))
            ((si-1 (a)) (in nil 1) (out a 1))
            ((si-0 ()) (in nil 3) (out nil 3))
            ((si-0 ()) (in a 1) (out a 1))
            ((si-0 ()) (in nil 1) (out nil 1))
            ((si-0 ()) (in c 2) (out nil 2))
            ((si-1 (c)) (in b 1) (out c 1))
            ((si-1 (b)) (in nil 2) (out b 2))
            ((si-0 ()) (in nil 1) (out nil 1)).
```

The purpose of this exercise is of course to check whether, given yet another input trajectory, its output trajectory can be "correctly" predicted using the information contained in the two training trajectories described above.

In particular, I shall use the following input trajectory:

```
in-traj-3:  (in c 0)
            (in a 1)
            (in b 6))
            (in nil 8))
```

with a given initial state and an assumption set, to determine what output trajectories can be predicted using my approach. Note that in the above pseudo-code representation, the third argument denotes the time point when the arrival event takes place, rather than a duration.

Since we know the internal structure of the system, we can of course perform a quantitative and completely deductive simulation, to determine the true IO trajectory for this system, given `in-traj-3` as its input trajectory. It is important to note that I am using the knowledge about the shipyard's internal structure as a baseline to measure the predictions of LDIR against it.

I shall assume that the shipyard is initially empty and that it follows a FIFO discipline. The pseudo-code representation of the correct IO trajectory is as follows:

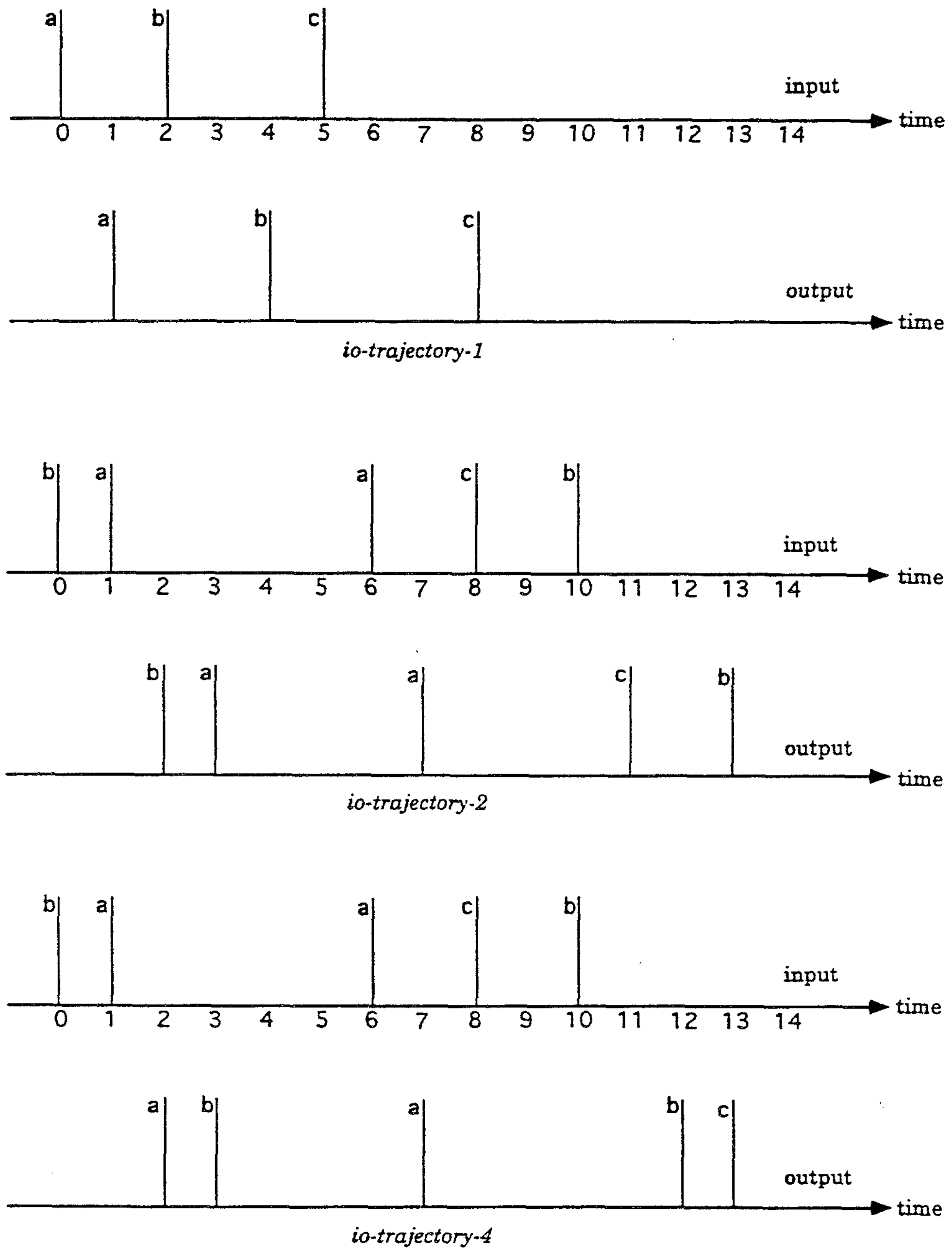
```
io-traj-3-fifo: ((si-0 ()) (in c 1) (out nil 1))
                ((si-1 (c)) (in a 2) (out c 2))
                ((si-1 (a)) (in nil 1) (out a 1))
                ((si-0 ()) (in nil 3) (out nil 3))
                ((si-0 ()) (in b 2) (out b 2))
                ((si-0 ()) (in nil 1) (out nil 1))
```

In the sequel, I shall use LDIR to predict a set of logically consistent IO trajectories for this input trajectory assuming no other knowledge about the system except for the two recorded IO trajectories, and I shall assess the fidelity of these hypothesized IO trajectories relative to the correct one shown above.

In a second step, I shall assume the true IO trajectory, *io-traj-3-fifo*, to be observed, adding its IO segments to the databases as additional evidence. Observation of *io-traj-3-fifo* may cause some of the previously hypothesized output trajectories, which had been predicted solely based on *io-traj-1* and *io-traj-2*, to become invalid, due to an inconsistency between previously made hypotheses and the new evidence. The truth value of these previously made hypotheses will thus change from TRUE to FALSE, and LDIR will need to revise its belief in the IO segments truth assignments to keep its databases fully consistent at all times.

6.2 Implementing DIR

I call the implementation of the discrete-event inductive reasoner a *Logic-based Discrete-event Inductive Reasoner (LDIR)*, since it is essentially a problem solver of the type described in Section 4.5. Before I discuss its highlights, a few words are



io-trajectory-1

io-trajectory-2

io-trajectory-4

Figure 6.1, Observed IO trajectories *io-traj-1*, *io-traj-2*, and *io-traj-4*.

in order about what parts of its implementation I am actually going to describe.

A problem solver (as described in [FdK93]) is designed such that its inference engine and its TMS can be discussed separately without much difficulty. I can discuss the inference engine part of the LDIR without requiring detailed discussion of the LTMS procedures (e.g., how the BCP works, Section 5.4) as long as I avoid referring to such details. The procedures of the LTMS tell us how its responsibilities are carried out. As this text would have to include more than a hundred additional pages to merely duplicate what is already available in [FdK93] about the LTMS and its implementation, I decided not to discuss its implementation at all, and refer the reader to the literature instead.

I have said that LDIR is a problem solver consisting of an inference engine and the LTMS. Forbus and de Kleer [FdK93] discuss in great detail implementations of several problem solvers based on different kinds of truth maintenance systems. For instance, they describe and discuss thoroughly the implementation of a problem solver called *Logic-based Tiny Rule Engine (LTRE)*, based on the *Logic-based TMS (LTMS)*.

The implementation of LDIR is embedded inside LTRE. Therefore, the inference engine of LDIR includes the capabilities described in Chapter 5 in addition to those of the inference engine of LTRE. Since LDIR does not make excessive use of LTRE's inference engine capabilities, I can exclude the discussion of its inference engine implementation without obscuring too much the discussion of LDIR's other features.

Given a discrete-event input trajectory and some discrete-event input/output trajectories, I recall that DIR, from a user's point of view, has to be able to (1) receive observed IO trajectories, (2) predict plausible discrete-event IO trajectories for some input trajectories, and (3) receive additional observed IO trajectories. I used the above ordering to indicate the obvious: at least some appropriate IO trajectories ought to be present before the DIR can be expected to predict output trajectories for any input trajectories. It is also possible that additional IO trajectories may become available after some IO trajectories have already been predicted.

The part of LDIR's inference engine that is beneficial to discuss is comprised of four "modules."

Add: has the responsibility to receive observed IO trajectories and to stash them away both in a database of its own and in the LTMS.

Consistency: contains the set of axioms (cf. Section 5.4.3.1) and procedures that examine consistency among IO segments.

Hypothesize: is the most elaborate module. Its main responsibility is to partition an input trajectory and predict its plausible output trajectories while insisting on consistency among all IO segments.

Inquiry: contains the user interface utilities as well as some other routines that are called upon by **Add**, **Consistency**, and **Hypothesize**.

I shall discuss only the first three of the modules since the **Inquiry** module contains only auxiliary routines that are generally self-explanatory.

First, I shall discuss **Add** without worrying about the consistency of the observed data yet — for now it is the responsibility of the user to supply consistent

IO trajectories. Then, I shall begin with the discussion of the **Consistency** module. Next, I complete the earlier discussion of the **Add** by describing how the procedures responsible for ensuring consistency can be used to automatically add only consistent IO segments. Finally I shall present a fairly extensive discussion of **Hypothesize** where all the major procedures are carried out under the watchful eyes of the LTMS. As the occasion arises, I shall also make use of procedures that belong to the **Inquiry** module.

6.2.1 Declaring Observed IO Trajectories

The **Add** module creates a structure (using `def-struct` in Common Lisp) containing slots for input events, output events, states, observed IO segments, hypothesized input segments, hypothesized output segments, LTRE, debugging, and a title. The slot for "debugging" is an auxiliary slot introduced for easier maintenance of the software. The slot for "LTRE" provides the link with the LTRE. The slot "title" is used for assigning a name to the entered structure that can be referred to in printouts. The slots "hypothesized input trajectory" and "hypothesized output trajectory" contain the partitioned input trajectory and its corresponding output trajectory. The remaining slots pertain to the sets defined for the iterative IOFO specification (cf. Section 5.2).

input events:	X
output events:	Y
states:	S
observed IO segments:	$IOspace_G$

The main procedure of this module is `add-io-traj(io-traj)`. It adds new IO segments to the observed IO segment slot called `io-space-g`, which is a simple hash-table. The procedure `get-out-seg-g(in-seg-i)` finds the Complete output segment `out-seg-f` for a Complete input segment `in-seg-i`. Of course, the module **Add** with the above procedures only supports storage and retrieval of IO segments to and from the hash-table. For example, given *io-traj-2* for the shipyard example described above, then issuing the command `add-io-traj(io-traj-2)` would enter the first IO segment as:

```
(((SI-0 ()) (IN B 1)) ((SI-1 (B)) (OUT NIL 1))).
```

The procedure `add-io-traj(io-traj)` iterates over the entire IO trajectory by adding each IO segment to `io-space-g` using the procedure

```
add-io-space-g(in-seg, out-seg, i-state, f-state).
```

It creates Complete input/output segments from an input segment, an output segment, an initial state and a final state. Then, for each observed IO trajectory, its corresponding IO segments are stashed in the hash-table `io-space-g` indexed in accordance with their initial states. Of course, the function `add-io-space-g` does not check for consistency among IO segments. In fact, since a hash-table is being used, if two IO segments have the same input segment and initial states yet different output segments, then whichever is stored last overrides the previous one!

6.2.2 Enforcing Consistency Among IO Segments

The **Consistency** module contains two procedures

1. `ensure-consistency-asn(cand, asn-set)`
2. `ensure-consistency-ast(cand)`

where asn and ast are abbreviations for assumption and assertion, respectively. cand refers to a candidate IO segment, either a predicted IO segment (an assumption) or an observed IO segment (an assertion) (cf. Section 5.4.3.1). Consequently, cand refers to a predicted IO segment in the former procedure, whereas it references an observed IO segment in the latter procedure. As I discussed in the previous chapter, the role of these procedures is to enforce consistency among all IO segments (observed and predicted).

The procedure ensure-consistency-asn(cand, asn-set) begins by collecting all IO segments that have the same initial state as the candidate IO segment from the LTMS into a set called nodie. (Note that all IO segments stashed in io-space-g are also known to the LTMS.) Then, if nodie is empty, it simply sends the predicted IO segment into the LTMS as a hypothesis with assumption asn-set, since there exists no previous IO segment that can possibly contradict it. However, if the set nodie is not empty, then the predicted IO segment has to be examined against all of its members. As I discussed in Section 5.4.3.1, the set nodie can contain both hypothesized and asserted IO segments. A procedure called check-consistency-asn(cand, nodie, asn-set) enforces the axioms that test whether the predicted IO segment is consistent with the IO segments contained in the set nodie. After the examination of the predicted IO segment against all IO segments in the set nodie, the newly predicted IO segment becomes a hypothesis (with proper assumption provided by asn-set) whenever no contradiction has been discovered.

However, if the predicted IO segment is inconsistent with at least one of the existing observed IO segments, it will be rejected. For example, if we have the predicted IO segment:

```
(((SI-1 (C)) (IN A 1)) ((SI-1 (A)) (OUT C 1)))
```

and `nodie` contains the asserted IO segment:

```
(((SI-1 (C)) (IN A 2)) ((SI-1 (A)) (OUT C 2)))
```

then the hypothesized IO segment is rejected.

Note that the above example is an interesting one. Remember that in this representation, I chose to abstract away the remaining time to complete a transaction (or the elapsed time since the beginning of the transaction, which contains the same information). Remember that vessels of type "C" require three time units for repair. The above *assertion* indicates that a vessel of type "A" arrived one time unit into the service of a vessel of type "C." Consequently, vessel "C" needs two more time units before it can depart from the shipyard. Thus, the assertion represents a logically meaningful IO segment. The above *prediction* indicates that a vessel of type "A" arrived two time units into the service of a vessel of type "C." Consequently, vessel "C" needs one more time unit before departure. Thus, the prediction also represents a logically meaningful IO segment. Yet, it will be rejected by LDIR as inconsistent.

Logic-based non-monotonic reasoners can commit two types of errors. Lacking complete knowledge, they can either accept a wrong piece of information as correct, or they can reject a correct piece of information as invalid. We already knew that overabstraction can lead to errors of type I, but the above example shows that it can also lead to errors of type II.

The problem could have been avoided easily by adding the time to completion as an additional piece of information to the initial state of each IO segment. In the above example, this would have led to the assertion:

```
(((SI-1 ((C 2))) (IN A 2)) ((SI-1 ((A 1))) (OUT C 2)))
```

whereas the prediction would have taken the form:

```
(((SI-1 ((C 1))) (IN A 1)) ((SI-1 ((A 1))) (OUT C 1)))
```

Then, the two initial states would have been different, and the two IO segments could have co-existed in the LTMS.

So, why did it choose to abstract away the time to completion if this was evidently a dumb thing to do? It is not very exciting to watch a non-monotonic reasoner perform on a system with complete information. It is much more interesting to see how the reasoner performs under conditions of incomplete information, how it makes wild assumptions, discovers new problems, struggles, backtracks, revises previous assumptions made, etc. Thus, withholding an essential piece of information, such as the time to completion, turns out to be much more instructive, and this is why we decided to "play dumb."

It is also possible that the predicted IO segment is inconsistent with an already existing hypothesized IO segment. What does `ensure-consistency-asn` do in this situation? It can do a couple of things, as described in Section 5.4.3.1. One possibility would be to retain the existing hypothesized IO segment while rejecting the newly predicted IO segment. Another choice would be the reverse. It would also be possible to reject the predicted IO segment and revise the truth value of the existing hypothesized IO segment from TRUE to FALSE. We simply lack the knowledge necessary to make an informed decision. Therefore, a more appropriate way seems to request additional knowledge to settle the dispute between the two competing hypothesized IO segments whenever possible. In the present implementation of LDIR, the user encounters a *break-point* indicating what the conflicting IO segments are and asking the user for further instructions.

Analogously, `ensure-consistency-ast(cand)` also determines the set of IO segments that have the same initial state as the candidate IO segment. When the set `nodie` is empty, the observed IO segment is asserted — again there can exist no contradictory IO segment. If, however, the set is not empty, then the function `check-consistency-ast(cand, nodie)` is called. This function has two temporary variables, both holding hypothesized IO segments. One holds hypothesized IO segments that are affirmed by the observed IO segment. For example:

```
(((SI-0 NIL) (IN C 1)) ((SI-1 (C)) (OUT NIL 1)))
```

might be a hypothesized IO segment. Then given the observed IO segment

```
(((SI-0 NIL) (IN C 2)) ((SI-1 (C)) (OUT NIL 2)))
```

the above hypothesized IO segment is converted to an assertion and also added to `io-space-g` using the `add-io-space-g` procedure.

The other temporary variable contains those IO segments that are violated given the observed IO segment. An example may be helpful. Suppose we already have the hypothesized IO segment:

```
(((SI-1 (C)) (IN A 1)) ((SI-0 NIL) (OUT A 1)))
```

which we know can't be true since we understand the internal structure of the system — the vessel "C" just "sank" in the shipyard (an error of type I), and we receive the newly observed IO segment:

```
(((SI-1 (C)) (IN A 2)) ((SI-1 (A)) (OUT C 2))).
```

In this case, the belief in the hypothesized IO segment has to be reversed, i.e., LDIR should assign to it a truth value of FALSE. The procedure iterates over all IO segments in the set `nodie` and updates the truth values of hypothesized IO segments as necessary. At the end, the hypothesized IO segments in the set `nodie` are either affirmed (converted to assertions), or left unchanged. What happens when an IO segment from `nodie` is a conflicting assertion? The procedure enters a *break-point*, displaying the two inconsistent IO segments. Again, the LTMS would become inconsistent with the addition of the observed IO segment as an assertion, and since not enough knowledge is currently available to decide the issue, the user is being asked for help. Note that, when dealing with observed IO segments, we do not use the `asn-set`.

Before I can resume my earlier discussion of the **Add** module, I need to consider one more procedure:

```
assert-ios!(in-seg-i, out-seg-f).
```

The procedure `assert-ios!` creates a candidate IO segment of the form `io-seg-k` (cf. Section 5.4.1) from a Complete input segment `in-seg-i` and a Complete output segment `out-seg-f`. Then, if this IO segment already exists as an assertion, it is detected as a repeat assertion, and the user is notified. If it exists as a hypothesis, then the former hypothesis is affirmed (converted to an assertion.) In case the IO segment has not be observed before, it is asserted through `ensure-consistency-ast(cand)`.

Now, with the aid of the **Consistency** module, the procedure `add-io-space-g` can not only include the observed IO segments, it can also test for potential contradictions between a newly observed IO segment and all previously observed IO segments. Given *io-traj-2*, the `add-io-traj(io-traj-2)` uses `assert-ios!(((si-0 ()) (in a 1)) ((si-0 ()) (in a1)))` to assert the first IO segment. Once this is successfully done (i.e., the observed IO segment has been added to the LTMS), the procedure `add-io-space-g(in-seg, out-seg, i-state, f-state)` is called to add the observed IO segment also to `io-space-g`. This procedure iterates over the entire IO trajectory. Now the LTMS contains all observed IO segments as propositions with their assigned truth values set to TRUE as discussed in the previous chapter. Again, note that these IO segments are also available in the hash-table `io-space-g` without their truth values.

6.2.3 Hypothesizing IO Segments for Unobserved Input Segments

Next, I discuss the remaining module **Hypothesize**. The top-level procedure in this module predicts output trajectories for a given input trajectory, `in-traj`, an initial state, `init-state`, and the assumption set, `asn-set`:

```
pred-out-traj(in-traj, init-state, asn-set).
```

Before I proceed any further, I recall that the assumption set `asn-set` has two elements. The first element can be either `longest`, `exact`, or `all` (cf. Section 5.4.4). The second element can be either `abs-length`, `abs-input`, or `abs-state` (cf. Section 5.3).

At the outset of a session, a "candidate" Complete input segment is determined based on partitioning, as discussed in Section 5.4.4, using `form-in-seg-i(in-traj, init-state)`. It creates a Complete input segment `in-seg-i` from the input trajectory `in-traj` and the initial state. Given the input trajectory:

```
in-traj:  (in c 0)
          (in a 1)
          (in b 6)
          (in nil 8))
```

and the initial state `(si-0 ())`, the first candidate Complete input segment is:

```
((SI-0 NIL) (IN C 1)).
```

The length of the candidate input segment is initially selected as the length between the first input event and the next input event. Thus, the second candidate input segment is:

```
((SI-1 (C)) (IN A 5))
```

where the initial state `(si-1 (c))` is the final state associated with the previously asserted or hypothesized IO segment. Once a candidate input segment is determined, three cases are possible. I discuss each in turn.

Case 1: In the simplest situation, which there exists an exact match between the candidate input segment and the input segment of an observed IO segment in the inference engine's database `io-space-g`. For example, given the candidate input segment:

```
(((SI-0 NIL) (IN B 2))
```

and the asserted IO segment:

```
(((SI-0 NIL) (IN B 2)) ((SI-0 NIL) (OUT B 2)))
```

the procedure `in-seg-i?(in-seg-i, init-state)` determines that there exists an exact match, i.e., the input segment of the asserted IO segment is identical to the candidate input segment. Now, the input segment and the output segment of this IO segment are appended to two variables holding retrieved/hypothesized IO segments (e.g., `Input-Trajectory-I` and `Output-Trajectory-I`, cf. Section 6.3.1) using the procedures `parse-in-traj` and `parse-out-traj`, respectively. These variables are lists containing Complete input segments and Complete output segments, respectively. To start the next round, the final state of the IO segment just hypothesized is computed, to be used for the initial state of the new input trajectory. The new input trajectory is the prior input trajectory from which the input segment — for which the corresponding IO segment was just found — has been chopped off. Now, `pred-out-traj` starts over again with the new input trajectory, the new initial state, and the original assumption set (the assumption set remains unchanged during the entire session.)

When Case 1 is not applicable, `pred-out-traj` has to resort to abstractions (cf. Section 5.4). I consider the following two situations separately:

$$\begin{aligned}\ell(\text{candidate input segment}) &= 1\\ \ell(\text{candidate input segment}) &> 1.\end{aligned}$$

When the candidate input segment is of unit length, only `elm-2` needs to be taken into account, otherwise both `elm-1` and `elm-2` are needed.

Case 2: The main procedure find-pred-io-1(in-seg-i, asn-set) attempts to hypothesize an IO segment for a candidate input segment in-seg-i of unit length (i.e., $\ell($in-seg-i$) = 1$), if such an IO segment can be predicted. First, it attempts to find all the IO segments that have the same input event and initial state as in-seg-i while ignoring their durations. If no such IO segment can be found with the specified input event and initial state, then, depending on the type of abstraction specified by the assumption set (input-equivalence or state-equivalence), one or more IO segments may be predicted using a procedure called find-io-pred-seg. These predicted IO segments are yet to be examined for consistency before they can become hypotheses.

Before I continue, I shall describe how the procedure assume-ios! works. It is similar to assert-ios! discussed in the previous section with the exception that it takes the additional argument asn-set. The procedure assume-ios! creates a candidate IO segment of the form io-seg-k (cf. Section 5.4.1) for both the Complete input segment in-seg-i and the Complete output segment out-seg-f. Then, if the predicted IO segment already exists as either a hypothesis or an assertion, it is simply detected as a repeat. In case the newly predicted IO segment has not been assumed before, then it is assumed if ensure-consistency-asn(cand, asn-set) supports it. This IO segment, of course, is not stored in io-space-g where only observed IO segments are stored. Just as in the previous case, the Complete input and output segments of the consistently hypothesized IO segment are extracted and stored appropriately in variables holding hypothesized input/output segments.

The procedure rem-incons-asn(pred-io-seg) removes inconsistent IO segments from those obtained by the procedure find-io-pred-sing. When no state or input event is abstracted, then two possibilities exist. In the first case, the predicted IO segment is consistent and is assumed using `assume-ios!`. It is added to all other IO segments in the LTMS as a hypothesis. Here is an example. Given the candidate input segment:

```
((SI-0 NIL) (IN C 1))
```

and having the observed IO segment

```
(((SI-0 NIL) (IN C 2)) ((SI-1 (C)) (OUT NIL 2)))
```

then the following predicted IO segment is determined by length abstraction as:

```
(((SI-0 NIL) (IN C 1)) ((SI-1 (C)) (OUT NIL 1))).
```

When this predicted IO segment is consistent with all other IO segments (asserted and hypothesized), then it is assumed; i.e.,

```
(((SI-0 NIL) (IN C 1)) ((SI-1 (C)) (OUT NIL 1))) via ABS-LENGTH.
```

In the second case, the predicted IO segment is inconsistent with at least one of the IO segments known to the LTMS. Then, it becomes necessary to use `elm-2` of the assumption set `asn-set`. Now, for example, based on input abstraction, one or more IO segments might be predicted. Again, similar to what was done above, it could be that none of them is consistent. In this situation, the LDIR cannot do anything more! It reports that the existing data is insufficient. In the case that one or more of the predicted IO segments are consistent with the LTMS's IO segments, one is selected randomly and assumed. Once a predicted IO segment is assumed, its input and output segments are stored. Then, the new input trajectory, the initial state, and the assumption set are passed on to pred-out-traj for another round. Now I discuss the remaining case.

Case 3: The distinction between this case and the previous one is that the duration of the candidate input segment is greater than unit length. That is, form-in-seg-i produces a candidate input segment such that ℓ(candidate input segment) > 1. This case starts with calling the following procedure

```
find-pred-io-n(in-seg-i, asn-set).
```

Given the candidate input segment in-seg-i, there must have not been anything matching it. When elm-1: longest, the duration of the input segment is reduced (abstracted) successively by one unit until (a) an existing IO segment (hypothesized or asserted) is found that has the same initial state and input event as the input segment; or (b) no existing IO segment is found with the same initial state and input event as the input segment.

If no IO segment can be found with length abstraction alone (i.e., situation (b)), then the following procedure is called

```
(1)    find-in-seg-longest(in-seg-i, asn-set)
```

within the procedure find-pred-io-n to abstract input, initial-state, or both based on elm-2, while requiring the longest segment to be found (elm-1: longest.) Hence elm-1 and elm-2 are used simultaneously.

For example, suppose elm-2: abs-input. Now, one or more IO segments are predicted by abstracting both the input event and length. Suppose the candidate input segment is

```
((SI-1 (C)) (IN A 5)).
```

Then, suppose there does not exist any IO segment in the LTMS's database that matches this input segment by abstracting its length alone. There does not exist a single IO segment with input event A and initial state (si-1 (C)). Suppose we have the following IO segment

```
(((SI-1 (C)) (IN B 1)) ((SI-1 (B)) (OUT C 1))).
```

Therefore, the in-seg-i's duration is first reduced by one unit, which results in the new candidate input segment ((SI-1 (C)) (IN A 4)). This input segment has the same initial state as the existing IO segment above. Then since the input is to be abstracted, there exists a match if we also abstract in-seg-i length according to elm-1: longest. Since the longest duration is asked for, the following IO segment is predicted

```
(((SI-1 (C)) (IN A 4)) ((SI-1 (B)) (OUT C 4))).
```

Note that the length reduction by one time unit would not have been absolutely necessary. This is a heuristic built into the current implementation of LDIR. If the selected abstraction is elm-1: longest and if LDIR must make use of input or state abstraction as well in order to find a match, it returns the longest left subsegment that does not lead to a contradiction, but never the original segment itself.

The reader may notice that input abstraction was much more harmful than length abstraction. Understanding the structure of the system, we know about some conservation principles. We know that what goes in must eventually come out again, and that nothing can ever come out of the shipyard that hasn't gone in earlier — to be enforced by some sort of continuity equation.

LDIR, being entirely based on behavioral knowledge and induction, doesn't know any such thing. The evidence gathered through *io-traj-1* and *io-traj-2* alone is insufficient to support continuity. Consequently, the "repair" of vessel "A" above involves a major remodeling — vessel "A" is evidently being remodeled to become a vessel of type "B."

Yet, there is nothing fundamentally wrong with the way LDIR reasons. There is no such thing as a free lunch. Either LDIR is provided with enough data to make sensible decisions, or it will have to engage in wild guesses — reasonable or not — in order not to give up.

The predicted IO segment and the above existing IO segment are consistent. Consequently, the "remodeling job" is accepted as a valid hypothesis. However, a so predicted IO segment can be in contradiction with existing evidence, in which case the prediction would have been rejected.

When `elm-1: exact`, the procedure

```
(2)    find-in-seg-exact(in-seg-i, asn-set)
```

is called. Once again, this procedure attempts to abstract the input segment's length only. As in the previous case, two situations, (a) and (b), are possible. I give an example. For the same candidate input segment as above (i.e., `((SI-1 (C)) (IN A 5)))`, we suppose there does not exist any IO segment in the database of LTMS that matches it through abstraction of its length alone. Having the same existing IO segment as above together with `elm-2: abs-input`, the candidate input segment is changed, by reducing its duration, to:

```
((SI-1 (C)) (IN A 4)).
```

Unfortunately, there is still no `exact` match even if we allow the input event to be abstracted. To this end, we shall need to shorten the duration further, until it matches that of the only existing IO segment with the correct initial state. The predicted IO segment is:

```
(((SI-1 (C)) (IN A 1)) ((SI-1 (B)) (OUT C 1))),
```

i.e., another "remodeling job."

A similar discussion can also be given for the remaining case — `elm-1: all`. There exists a third procedure:

(3) `find-in-seg-all(in-seg-i, asn-set)`

to handle this type of abstraction. Additional scenarios are also possible by selecting one of the three length abstraction types, as well as abstracting the input event and the initial state simultaneously. I shall not discuss them here.

At the end of any of the above three procedures, a set of predicted IO segments is returned. As in the previous case, it is possible that any or all of them might be inconsistent. The procedure `rem-incons-asn` is called upon to remove those that are inconsistent. If multiple IO segments are predicted and either `elm-1: exact` or `longest`, then one of them is selected (either randomly or based on some a priori knowledge) and assumed if consistent. When `elm-1: all`, then all IO segments are assumed that do not contradict any of the existing IO segments. This can lead to multiple predicted IO trajectories. It is also possible that length, input event, and initial state abstractions do not lead to hypothesized IO segment(s). If this happens, the user is notified, and the procedure `pred-out-traj` comes to a halt.

When an IO segment is predicted and hypothesized for a candidate input segment, then the procedure `pred-out-traj` is called once upon again with the remainder of the input trajectory, the final state of the just computed hypothesized IO segment used as the new initial state (for reasons of continuity), and the selected assumption set-II. Of course, before calling the procedure `pred-out-traj`, the respective input and output segments of the hypothesized IO segments are appended to those previously computed using the procedures `parse-in-traj` and `parse-out-traj`.

6.3 Sample Scenarios for the Shipyard Example

I have used some fragments of *io-traj-1*, *io-traj-2*, and *in-traj-3* to illustrate the functioning of the **Add**, **Consistency**, and **Hypothesize** modules. Now, I present four scenarios that share the same input trajectory and initial state, but use distinct assumption sets. I shall first present each individually. Then, I shall collect their predicted IO trajectories and discuss them collectively. In each of these scenarios, I assume that the LDIR has already processed the IO trajectories *io-traj-1* and *io-traj-2*, and thus has knowledge of the following observed IO segments.

```
 1.  (((SI-1 (B)) (IN NIL 2)) ((SI-0 ()) (OUT B 2))):  TRUE
 2.  (((SI-1 (C)) (IN B 1)) ((SI-1 (B)) (OUT C 1))):  TRUE
 3.  (((SI-1 (A)) (IN NIL 1)) ((SI-0 ()) (OUT A 1))):  TRUE
 4.  (((SI-1 (B)) (IN A 1)) ((SI-1 (A)) (OUT B 1))):  TRUE
 5.  (((SI-0 ()) (IN C 2)) ((SI-1 (C)) (OUT NIL 2))):  TRUE
 6.  (((SI-0 ()) (IN NIL 3)) ((SI-0 ()) (OUT NIL 3))):  TRUE
 7.  (((SI-0 ()) (IN B 1)) ((SI-1 (B)) (OUT NIL 1))):  TRUE
 8.  (((SI-0 ()) (IN C 3)) ((SI-0 ()) (OUT C 3))):  TRUE
 9.  (((SI-0 ()) (IN B 2)) ((SI-0 ()) (OUT B 2))):  TRUE
10.  (((SI-0 ()) (IN NIL 1)) ((SI-0 ()) (OUT NIL 1))):  TRUE
11.  (((SI-0 ()) (IN A 1)) ((SI-0 ()) (OUT A 1))):  TRUE
```

6.3.1 Scenario 1

In this example, the procedure pred-out-traj called upon with the initial state, assumption set, and input trajectory:

```
initial-state:  (si-0 ())
asn-set:  (longest abs-input)
in-traj-3:  (in c 0)
            (in a 1)
            (in b 6)
            (in nil 8))
```

predicts some IO segments, examines them for consistency, and generates a pair of consistent input and output trajectories. I present the results generated by LDIR with some cosmetic changes for better readability. I also provide explanations as needed.

```
pred-out-traj(in-traj-3 (initial-state asn-set))
candidate input seg:  ((SI-0 ()) (IN C 1))
candidate input seg ((SI-0 ()) (IN C 1)) does not exist.
```

Candidate IO segments with matching input event and initial state are computed using `find-hyp-io-1` since the candidate input segment is of unit length. We have:

```
(((SI-0 ()) (IN C 2)) ((SI-1 (C)) (OUT NIL 2)))
(((SI-0 ()) (IN C 3)) ((SI-0 ()) (OUT C 3))).
```

These IO segments are found based on Case 2. Although two IO segments are found, only one of them leads to a hypothesized IO segment. We obtain the following predicted IO segment:

```
(((SI-0 ()) (IN C 1)) ((SI-1 (C)) (OUT NIL 1))).
```

At this point, since the predicted IO segment does not violate any of the existing IO segments, it is assumed and becomes a hypothesis. Note that only the length has been ignored (abstracted). That is:

```
hypo-io-seg:  (((SI-0 ()) (IN C 1)) ((SI-1 (C)) (OUT NIL 1)))
via ABS-LENGTH.
```

Comparing the hypothesized IO segment with the true first IO segment of *io-traj-3-fifo*, we find that length abstraction has worked beautifully. LDIR has indeed hypothesized the correct IO segment in this case.

Then, a new input trajectory is computed by cutting off the first segment before calling `pred-out-traj` once again with the same assumption set, yet with a different initial state. The initial state is the final state of the previous IO segment. Hence:

```
initial-state:  (si-1 (c))
```

```
 in-traj-3:  (in a 1)
             (in b 6)
             (in nil 8))

input Traj:  ((IN A 1) (IN B 6) (IN NIL 8))

candidate input seg:  ((SI-1 (C)) (IN A 5))

candidate input seg ((SI-1 (C)) (IN A 5)) does not exist.
```

The procedure in-seg-i? does not find any IO segment in the database having an input segment that matches the candidate input segment completely (i.e., they differ at least in either the initial state, the input event, or the duration.) It then attempts to abstract the duration of the candidate's input segment. This time, Case 3 comes to bear, since the length of the input segment is greater than one. Hence the procedure find-hyp-io-n is called upon. Also, since the assumption set asks for the "longest" match, the procedure find-in-seg-longest is called to partition the candidate input segment. Consequently, LDIR finds the longest possible IO segments for which both the length and the input event (if necessary) are abstracted by partitioning the input segment (IN A 5). We obtain:

```
reduced input seg:  ((SI-1 (C)) (IN A 4))

cand IO seg:  (((SI-1 (C)) (IN B 1)) ((SI-1 (B)) (OUT C 1)))

pred IO seg:  (((SI-1 (C)) (IN A 4)) ((SI-1 (B)) (OUT C 4)))

hypo IO seg:  (((SI-1 (C)) (IN A 4)) ((SI-1 (B)) (OUT C 4)))
via ABS-LENGTH & ABS-INPUT.
```

thence one of these "remodeling jobs" our shipyard is already so famous for. The longest consistent IO segment was found and assumed. Since the input event was ignored (abstracted), the input event for the hypothesized IO segment had to be changed from the model IO segment used in the abstraction in order to fit the bill, which led to violation of the conservation principle.

Evidently, it would have been more reasonable to change the final state in accordance with the abstracted input event to:

```
(((SI-1 (C)) (IN A 4)) ((SI-1 (A)) (OUT C 4))).
```

but LDIR had no way of knowing that. For LDIR, a state is a compact entity that cannot be analyzed internally for its semantic meaning. Consequently, LDIR did the best it could under the given circumstances.

Next, pred-out-traj is called upon with the following newly computed input trajectory and initial state:

```
 initial-state:  (si-1 (b))
 in-traj-3:  (in nil 5)
             (in b 6)
             (in nil 8))

input Traj:  ((IN NIL 5) (IN B 6) (IN NIL 8))

candidate input seg:  ((SI-1 (B)) (IN NIL 1))

candidate input seg ((SI-1 (B)) (IN NIL 1)) does not exist.
```

Time has advanced to unit 5 by now, yet the next arrival of a vessel (of type "B") is only scheduled for time point 6. Since the flow of time is a universal concept that doesn't depend on the situation to be modeled, LDIR is allowed to make use of deeper knowledge assuming the continuity of time. Hence LDIR is able to insert a NIL segment of length one into the input trajectory.

Since the new candidate input segment does not exist, procedure find-hyp-io-1 is called upon as the candidate input segment is of unit length. The following IO segment is predicted:

```
cand IO seg:  (((SI-1 (B)) (IN NIL 2)) ((SI-0 ()) (OUT B 2)))
pred IO seg:  (((SI-1 (B)) (IN NIL 1)) ((SI-1 (B)) (OUT NIL 1))).
```

The NIL event can be treated in a special way, since it is independent of the application at hand. Shortening a candidate IO segment leads necessarily to a NIL output event, and a IO segment with both NIL input and output events cannot possibly change its state. Thus, the above prediction is reasonable and can be made without drawing upon application-dependent structural knowledge. Hence we accept the following IO segment as a hypothesis once it has been examined and authorized by the consistency axioms.

```
hypo IO seg:  (((SI-1 (B)) (IN NIL 1)) ((SI-1 (B)) (OUT NIL 1)))
via ABS-LENGTH.
```

Now the remaining input segment is (IN B 2) with initial state (si-1 (b)).

```
 initial-state:  (si-1 (b))
 in-traj-3:  (in b 6)
             (in nil 8))

input Traj:  ((IN B 6) (IN NIL 8))
candidate input seg:  ((SI-1 (B)) (IN B 2))
candidate input seg ((SI-1 (B)) (IN B 2)) does not exist.
```

There does not exist an IO segment with initial state (SI-1 (B)) and candidate input segment (IN B 2). Abstracting the length alone is not sufficient, and it becomes necessary to also abstract the input event, i.e., find-in-seg-longest is called upon. The following consistent hypothesized IO segment is predicted:

```
(((SI-1 (B)) (IN B 1)) ((SI-1 (A)) (OUT B 1)))
via ABS-LENGTH & ABS-INPUT.
```

i.e., another remodeling job.

At this point in time, the new input trajectory and initial state become:

```
 initial-state:  (si-1 (a))
 in-traj-3:  (in nil 7)
             (in nil 8))

input Traj:  ((IN NIL 7) (IN NIL 8))
candidate input seg:  ((SI-1 (A)) (IN NIL 1))
candidate input seg ((SI-1 (A)) (IN NIL 1)) does exist.
```

Since a matching assertion can be found in the LTMS, the IO segment can be hypothesized without any abstraction.

```
(((SI-1 (A)) (IN NIL 1)) ((SI-0 ()) (OUT A 1))).
```

Now, `pred-out-traj` comes to a halt since the input trajectory has been successfully partitioned into input segments and corresponding output segments. The input and output trajectory pairs, depicted in Figure 6.2 on page 208, generated by LDIR are:

```
Input-Trajectory-I:  ((SI-0 ()) (IN C 1))
                     ((SI-1 (C)) (IN A 4))
                     ((SI-1 (B)) (IN NIL 1))
                     ((SI-1 (B)) (IN B 1))
                     ((SI-1 (A)) (IN NIL 1))

Output-Trajectory-I:  ((SI-1 (C)) (OUT NIL 1))
                      ((SI-1 (B)) (OUT C 4))
                      ((SI-1 (B)) (OUT NIL 1))
                      ((SI-1 (A)) (OUT B 1))
                      ((SI-0 ()) (OUT A 1)))
```

The top graph of Figure 6.2 is what the observer on the shore would see. It looks inconspicuous. The observer might hypothesize that the repair of vessel "C" required 5 time units. Thereafter, repair on vessel "A" was started, but this job got interrupted at time unit 6 by a higher priority job when vessel "B" arrived. Vessel "B" took one time unit for its repair, after which time the work on vessel "A" was resumed. Since the observer on the shore doesn't have access to the internal state information of the shipyard, he or she has no reason to assume foul play and would never guess that two major remodeling jobs have taken place.

At this point, procedure `find-IO-seg-asn` selects those IO segments that represent new hypotheses. The hypothesized IO segments are:

```
(((SI-1 (B)) (IN B 1)) ((SI-1 (A)) (OUT B 1)))
(((SI-1 (B)) (IN NIL 1)) ((SI-1 (B)) (OUT NIL 1)))
(((SI-1 (C)) (IN A 4)) ((SI-1 (B)) (OUT C 4)))
(((SI-0 ()) (IN C 1)) ((SI-1 (C)) (OUT NIL 1))).
```

We can query the LTMS for a list of all IO segments and their truth values. Beside from the previously available assertions, the LTMS now contains four additional hypothesized IO segments:

```
12. (((SI-1 (B)) (IN B 1)) ((SI-1 (A)) (OUT B 1))):  TRUE
13. (((SI-1 (B)) (IN NIL 1)) ((SI-1 (B)) (OUT NIL 1))):  TRUE
14. (((SI-1 (C)) (IN A 4)) ((SI-1 (B)) (OUT C 4))):  TRUE
15. (((SI-0 ()) (IN C 1)) ((SI-1 (C)) (OUT NIL 1))):  TRUE
```

Next, let us assume that the IO trajectory *io-traj-3-fifo* is being observed (cf. Section 5-1). It is partitioned into IO segments and added to the already available assertions by the `add-io-space-g` procedure. However, it is not possible to simply add the new observations unchecked, since they may contradict one or more of the already entered IO segments. Therefore, each new pair is tested for consistency, and if it is neither redundant with nor contradicts the previous evidence, it is added to the databases.

In the given example, the first IO segment:

```
(((SI-0 ()) (IN C 1)) ((SI-1 (C)) (OUT NIL 1)))
```

already exists as a hypothesis. In this case, the hypothesis is converted to an assertion. The second IO segment:

```
((SI-1 (C)) (IN A 2)) ((SI-1 (A)) (OUT C 2))
```

contradicts the previously made hypothesis:

```
(((SI-1 (C)) (IN A 4)) ((SI-1 (B)) (OUT C 4)))
```

which will now have to be retracted and its negation assumed. That is:

```
hypo IO seg:  (NOT (((SI-1 (C)) (IN A 4)) ((SI-1 (B)) (OUT C 4))))
via (((SI-1 (C)) (IN A 2)) ((SI-1 (A)) (OUT C 2))).
```

Once the negated IO segment is assumed, the observed IO segment is asserted.

```
observed IO seg:  (((SI-1 (C)) (IN A 2)) ((SI-1 (A)) (OUT C 2)))
via OBSERVED-DATA.
```

Of course, if a duplicate IO segment is found, it is noted and ignored. In the given example, the following three IO segments have already been asserted as IO segments through either *io-traj-1* or *io-traj-2*, and are thence ignored. That is:

```
duplicate :  (((SI-1 (A)) (IN NIL 1)) ((SI-0 ()) (OUT A 1)))
duplicate :  (((SI-0 ()) (IN NIL 3)) ((SI-0 ()) (OUT NIL 3)))
duplicate :  (((SI-0 ()) (IN B 2)) ((SI-0 ()) (OUT B 2))).
```

A query to the LTMS for its current contents shows the following IO segments and their truth values.

```
 1.  (((SI-1 (B)) (IN NIL 2)) ((SI-0 ()) (OUT B 2))):  TRUE
 2.  (((SI-1 (C)) (IN B 1)) ((SI-1 (B)) (OUT C 1))):  TRUE
 3.  (((SI-1 (A)) (IN NIL 1)) ((SI-0 ()) (OUT A 1))):  TRUE
 4.  (((SI-1 (B)) (IN A 1)) ((SI-1 (A)) (OUT B 1))):  TRUE
 5.  (((SI-0 ()) (IN C 2)) ((SI-1 (C)) (OUT NIL 2))):  TRUE
 6.  (((SI-0 ()) (IN NIL 3)) ((SI-0 ()) (OUT NIL 3))):  TRUE
 7.  (((SI-0 ()) (IN B 1)) ((SI-1 (B)) (OUT NIL 1))):  TRUE
 8.  (((SI-0 ()) (IN C 3)) ((SI-0 ()) (OUT C 3))):  TRUE
 9.  (((SI-0 ()) (IN B 2)) ((SI-0 ()) (OUT B 2))):  TRUE
10.  (((SI-0 ()) (IN NIL 1)) ((SI-0 ()) (OUT NIL 1))):  TRUE
11.  (((SI-0 ()) (IN A 1)) ((SI-0 ()) (OUT A 1))):  TRUE
12.  (((SI-1 (B)) (IN B 1)) ((SI-1 (A)) (OUT B 1))):  TRUE
13.  (((SI-1 (B)) (IN NIL 1)) ((SI-1 (B)) (OUT NIL 1))):  TRUE
14.  (((SI-1 (C)) (IN A 4)) ((SI-1 (B)) (OUT C 4))):  FALSE
15.  (((SI-0 ()) (IN C 1)) ((SI-1 (C)) (OUT NIL 1))):  TRUE
16.  (((SI-1 (C)) (IN A 2)) ((SI-1 (A)) (OUT C 2))):  TRUE
```

Note that `(((SI-1 (C)) (IN A 4)) ((SI-1 (B)) (OUT C 4)))` had a truth value of TRUE prior to adding *io-traj-3-fifo*. Also note that the IO segment:

```
(((SI-0 ()) (IN C 1)) ((SI-1 (C)) (OUT NIL 1))):  TRUE
```

is no longer a hypothesis, as it should not be! That is, the query `find-IO-seg-asn` returns only three IO segments as hypotheses at this time:

```
12. (((SI-1 (B)) (IN B 1)) ((SI-1 (A)) (OUT B 1))):  TRUE
13. (((SI-1 (B)) (IN NIL 1)) ((SI-1 (B)) (OUT NIL 1))):  TRUE
15. (((SI-1 (C)) (IN A 4)) ((SI-1 (B)) (OUT C 4))):  FALSE
```

It should be remarked that only a single assertion had been missing in the LTMS, namely:

```
16. (((SI-1 (C)) (IN A 2)) ((SI-1 (A)) (OUT C 2))):  TRUE
```

This one additional assertion would have sufficed to enable LDIR to predict the correct IO trajectory, given the LTMS and `in-traj-3`. Because of a lack of evidence, LDIR had to make a wild hypothesis. It did so using the available evidence in the best way possible, but came up short, suggesting an unfortunately incorrect hypothesis that then led to further mishap down the road. Yet, it is important to point out that *very little additional knowledge* would have been needed to do the job right.

6.3.2 Scenario 2

The initial state and the input trajectory for this example remain the same as in the previous scenario. Only the assumption set is changed to `asn-set: (exact abs-input)`.

Since the first candidate input segment for this input trajectory did not require any input abstraction before, it won't require such an abstraction now either. Consequently, the first hypothesis remains the same, namely:

```
((SI-0 ()) (IN C 1)) ((SI-1 (C)) (OUT NIL 1)).
```

Next, a reduced input trajectory and a new initial state are computed as before, calling `pred-out-traj`. We have:

```
 initial-state:  (si-1 (c))
 asn-set:  (exact abs-input)
 in-traj-3:  (in a 1)
             (in b 6)
             (in nil 8)

candidate input seg:  ((SI-1 (C)) (IN A 5))
candidate input seg ((SI-1 (C)) (IN A 5)) does not exist.
```

Since the procedure `in-seg-i?` fails and since the duration of the candidate input segment is greater than unit length, the procedure **`find-hyp-io-n`** is called in accordance with Case 3. Given the assumption set's first element, **`elm-1: exact`**, procedure **`find-in-seg-exact`** is called to determine the predicted IO segments. The duration of the candidate IO segment is reduced successively until one or more desired IO segments are found. The first reduced input segment is:

```
reduced input seg:  ((SI-1 (C)) (IN A 4))
```

for which no desired IO segments can be found. Consequently, the candidate input segment will have its duration reduced further. For the following two reduced input segments still no desired IO segments can be predicted.

```
reduced input seg:  ((SI-1 (C)) (IN A 3))
reduced input seg:  ((SI-1 (C)) (IN A 2)).
```

Finally, even for the reduced candidate input segment `((SI-1 (C)) (IN A 1))`, no candidate IO segment can be found based on abstracting the length alone. Hence, the input event abstraction becomes necessary. Again, the candidate input segment had to be reduced to `((SI-1 (C)) (IN A 1))` before the following IO segment could be found:

```
(((SI-1 (C)) (IN B 1)) ((SI-1 (B)) (OUT C 1))).
```

that has a different input event, but the correct initial state and the correct (exact) duration. Consequently, the following IO segment will be predicted:

```
pred IO seg:  (((SI-1 (C)) (IN A 1)) ((SI-1 (B)) (OUT C 1))),
```

again suggesting a "remodeling job."

Since this prediction does not contradict any existing IO segment, it is assumed. The procedure `assume-ios!` is called to make the predicted IO segment a hypothesis.

The next reduced input trajectory and its initial state are:

```
initial-state:  (si-1 (b))
in-traj-3: (in nil 2)
           (in b 6)
           (in nil 8).
```

The candidate input segment is thus `((SI-1 (B)) (IN NIL 4))`. Also, since no IO segment can be found for this candidate input segment, its duration is reduced until an IO segment is predicted for it. For the shortened candidate input segment `((si-1 (b)) (in nil 2))`, we find a matching IO segment without need for input abstraction. The predicted IO segment is thus:

```
(((SI-1 (B)) (IN NIL 2)) ((SI-0 ()) (OUT B 2))).
```

As indicated previously, the predicted IO segment is detected as an existing IO segment (an assertion); therefore, it must be consistent with the existing IO segments. Hence there is no need for further consistency checking.

I continue with:

```
initial-state:  (si-0 ())
in-traj-3:  (in nil 4)
            (in b 6)
            (in nil 8).
```

The next input segment is `((SI-0 ()) (IN NIL 2))`. This input segment does not exist and thus abstraction on its length is attempted first. The reduced input segment, `((si-0 nil) (in nil 1))`, leads to the following redundant assertion:

```
(((SI-0 ()) (IN NIL 1)) ((SI-0 ()) (OUT NIL 1))).
```

The next reduced input trajectory and its initial state are:

```
initial-state:  (si-0 ())
in-traj-3:  (in nil 5)
            (in b 6)
            (in nil 8).
```

The following IO segment is found in `io-space-g` for the input segment `((SI-0 ()) (IN NIL 1))` without any abstraction:

```
(((SI-0 ()) (IN NIL 1)) ((SI-0 ()) (OUT NIL 1))).
```

The last reduced input trajectory and initial state are:

```
initial-state:  (si-0 ())
in-traj-3:  (in b 6)
            (in nil 8)
```

leading to the input segment `((SI-0 ()) (IN B 2))`, for which the following IO segment can be predicted:

```
(((SI-0 ()) (IN B 2)) ((SI-0 ()) (OUT B 2)))
```

without any need of abstraction.

Partitioning the input trajectory and computing its IO segments results in the predicted input and output trajectories given below and shown in Figure 6.2.

```
Input-Trajectory-II:  ((SI-0 ()) (IN C 1))
                      ((SI-1 (C)) (IN A 1))
                      ((SI-1 (B)) (IN NIL 2))
                      ((SI-0 ()) (IN NIL 1))
                      ((SI-0 ()) (IN NIL 1))
                      ((SI-0 ()) (IN B 2))

Output-Trajectory-II:  ((SI-1 (C)) (OUT NIL 1))
                       ((SI-1 (B)) (OUT C 1))
                       ((SI-0 ()) (OUT B 2))
                       ((SI-0 ()) (OUT NIL 1))
                       ((SI-0 ()) (OUT NIL 1))
                       ((SI-0 ()) (OUT B 2)))
```

This time, an observer on the shore would have been able to discover the remodeling job, since one vessel of type "B" went in, but two vessels of type "B" came out.

We can query the LTMS to find those IO segments that represent assumptions. They are:

```
12. (((SI-1 (C)) (IN A 1)) ((SI-1 (B)) (OUT C 1))):  TRUE
13. (((SI-0 ()) (IN C 1)) ((SI-1 (C)) (OUT NIL 1)):  TRUE
```

Using the assumption set `asn-set: (exact abs-input)`, we ended up with fewer hypothesized IO segments, and input abstraction had to be used less frequently. However, the input trajectory is partitioned into more input segments, and there is no reason to believe, just on the basis of what LDIR knows, that the predictions are any better or worse than in Scenario 1.

Next, we assume as in Scenario 1 that IO trajectory *io-traj-3-fifo* is now being observed. It is partitioned into IO segments and added to the databases by the `add-io-space-g` procedure. Again, each new IO segment must be checked for consistency before it can be added. The steps taken in adding the newly observed IO segments to the already existing IO segments are to a large extent the same as those illustrated in Scenario 1. The main difference is the second observed IO segment.

```
((SI-1 (C)) (IN A 2)) ((SI-1 (A)) (OUT C 2))
```

contradicts the assumption:

```
(((SI-1 (C)) (IN A 1)) ((SI-1 (B)) (OUT C 1))).
```

Therefore, this hypothesis must be retracted and its negation assumed instead.

Hence the following IO segment:

```
(NOT (((SI-1 (C)) (IN A 1)) ((SI-1 (B)) (OUT C 1))))
assumed via (((SI-1 (C)) (IN A 2)) ((SI-1 (A)) (OUT C 2))).
```

Once the negated IO segment is assumed, the observed IO segment is asserted as before:

```
Asserted (((SI-1 (C)) (IN A 2)) ((SI-1 (A)) (OUT C 2)))
via OBSERVED-DATA
```

Again, duplicate IO segments are noted and ignored. Once the databases have fully absorbed the newly available information, the following IO segments with their truth values can be queried from the LTMS:

```
 1. (((SI-1 (B)) (IN NIL 2)) ((SI-0 ()) (OUT B 2))): TRUE
 2. (((SI-1 (C)) (IN B 1)) ((SI-1 (B)) (OUT C 1))): TRUE
 3. (((SI-1 (A)) (IN NIL 1)) ((SI-0 ()) (OUT A 1))): TRUE
 4. (((SI-1 (B)) (IN A 1)) ((SI-1 (A)) (OUT B 1))): TRUE
 5. (((SI-0 ()) (IN C 2)) ((SI-1 (C)) (OUT NIL 2))): TRUE
 6. (((SI-0 ()) (IN NIL 3)) ((SI-0 ()) (OUT NIL 3))): TRUE
 7. (((SI-0 ()) (IN B 1)) ((SI-1 (B)) (OUT NIL 1))): TRUE
 8. (((SI-0 ()) (IN C 3)) ((SI-0 ()) (OUT C 3))): TRUE
 9. (((SI-0 ()) (IN B 2)) ((SI-0 ()) (OUT B 2))): TRUE
10. (((SI-0 ()) (IN NIL 1)) ((SI-0 ()) (OUT NIL 1))): TRUE
11. (((SI-0 ()) (IN A 1)) ((SI-0 ()) (OUT A 1))): TRUE
12. (((SI-1 (C)) (IN A 1)) ((SI-1 (B)) (OUT C 1))): FALSE
13. (((SI-0 ()) (IN C 1)) ((SI-1 (C)) (OUT NIL 1))): TRUE
14. (((SI-1 (C)) (IN A 2)) ((SI-1 (A)) (OUT C 2))): TRUE
```

After the addition of *io-traj-3-fifo*, the query `find-IO-seg-asn` returns a single hypothesized IO segment only:

```
12. (((SI-1 (C)) (IN A 1)) ((SI-1 (B)) (OUT C 1))): FALSE
```

as was to be expected. This is very encouraging indeed. Hypotheses are a necessary evil in non-monotonic reasoning. They enable the reasoner to proceed under conditions of incomplete knowledge. Yet, it is encouraging to see that few hypotheses were necessary in the first place, and that the only remaining hypothesis at this point has been deactivated for all practical purposes by being negated.

6.3.3 Scenario 3

I only present the features that are specific to this scenario. This scenario is used to illustrate the use of state equivalence. Hence we have the same input trajectory and initial state, but the assumption set is `asn-set: (exact abs-state)`.

```
initial-state:  (si-0 ())
asn-set:  (exact abs-state)
in-traj-3:  (in c 0)
            (in a 1)
            (in b 6)
            (in nil 8)
```

The first prediction must evidently be the same as in the earlier scenarios. However, already in the second prediction, state abstraction comes to bear. We start out with the following situation:

```
initial-state:  (si-1 (c))
in-traj-3:  (in a 1)
            (in b 6)
            (in nil 8))

input Traj:  ((IN A 1) (IN B 6) (IN NIL 8))

candidate input seg:  ((SI-1 (C)) (IN A 5))

candidate input seg ((SI-1 (C)) (IN A 5)) does not exist.
```

Length abstraction alone won't do the trick. Since `elm-1: exact`, LDIR reduces the length until it finds a perfect match of input event and duration. It finds two candidate IO segments in the LTMS:

```
 4. (((SI-1 (B)) (IN A 1)) ((SI-1 (A)) (OUT B 1)))
11. (((SI-0 ()) (IN A 1)) ((SI-0 ()) (OUT A 1)))
```

leading to the possible predictions:

```
(((SI-1 (C)) (IN A 1)) ((SI-1 (A)) (OUT B 1)))
(((SI-1 (C)) (IN A 1)) ((SI-0 ()) (OUT A 1))).
```

Since both predictions are consistent with all the entries currently in the LTMS, it picks one arbitrarily. LDIR decided on the second candidate, thus:

```
hypo IO seg:  (((SI-1 (C)) (IN A 1)) ((SI-0 ()) (OUT A 1)))
via ABS-LENGTH & ABS-STATE.
```

No remodeling job this time! LDIR is much more destructive now — vessel "C" is sunk in the shipyard.

The remainder of the predictions are straightforward. The predicted input and output trajectories are (see Figure 6.2):

```
Input-Trajectory-III:  ((SI-0 ()) (IN C 1))
                       ((SI-1 (C)) (IN A 1))
                       ((SI-0 ()) (IN NIL 3))
                       ((SI-0 ()) (IN NIL 1))
                       ((SI-0 ()) (IN B 2))

Output-Trajectory-III:  ((SI-1 (C)) (OUT NIL 1))
                        ((SI-0 ()) (OUT A 1))
                        ((SI-0 ()) (OUT NIL 3))
                        ((SI-0 ()) (OUT NIL 1))
                        ((SI-0 ()) (OUT B 2))).
```

Note that the partitioning of the input trajectory is different than in the previous two scenarios. Now, we can query the LTMS to find all IO segments.

```
 1. (((SI-1 (B)) (IN NIL 2)) ((SI-0 ()) (OUT B 2))):  TRUE
 2. (((SI-1 (C)) (IN B 1)) ((SI-1 (B)) (OUT C 1))):  TRUE
 3. (((SI-1 (A)) (IN NIL 1)) ((SI-0 ()) (OUT A 1))):  TRUE
 4. (((SI-1 (B)) (IN A 1)) ((SI-1 (A)) (OUT B 1))):  TRUE
 5. (((SI-0 ()) (IN C 2)) ((SI-1 (C)) (OUT NIL 2))):  TRUE
 6. (((SI-0 ()) (IN NIL 3)) ((SI-0 ()) (OUT NIL 3))):  TRUE
 7. (((SI-0 ()) (IN B 1)) ((SI-1 (B)) (OUT NIL 1))):  TRUE
 8. (((SI-0 ()) (IN C 3)) ((SI-0 ()) (OUT C 3))):  TRUE
 9. (((SI-0 ()) (IN B 2)) ((SI-0 ()) (OUT B 2))):  TRUE
10. (((SI-0 ()) (IN NIL 1)) ((SI-0 ()) (OUT NIL 1))):  TRUE
11. (((SI-0 ()) (IN A 1)) ((SI-0 ()) (OUT A 1))):  TRUE
12. (((SI-1 (C)) (IN A 1)) ((SI-0 ())) (OUT A 1))):  TRUE
13. (((SI-0 ()) (IN C 1)) ((SI-1 (C)) (OUT NIL 1))):  TRUE
```

Of these IO segments, the last two are hypotheses. As before, they can be obtained separately by using the query `find-io-seg-asn`.

Again, adding *io-traj-3-fifo* results in an additional IO segment:

```
14. (((SI-1 (C)) (IN A 2)) ((SI-1 (A)) (OUT C 2))):  TRUE
```

This observed IO segment also contradicts (and thus negates) the hypothesized IO segment #12, which is also the only remaining hypothesized IO segment given the three observed IO trajectories, thus a similar situation as in Scenario 2.

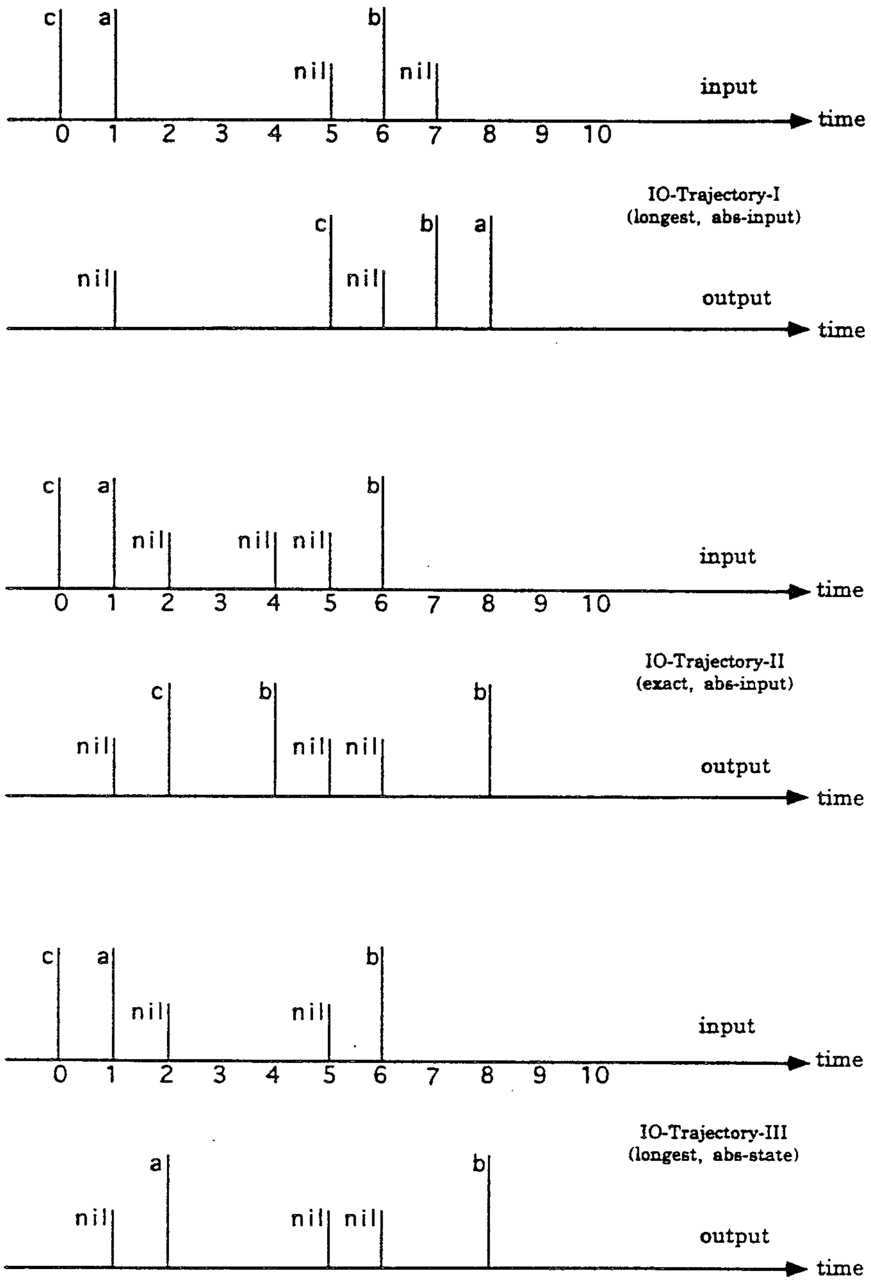


Figure 6.2, Predicted IO trajectories (FIFO Discipline): IO-Trajectory-I, IO-Trajectory-II, and IO-Trajectory-III

6.4 PR vs FIFO Disciplines

Let us now introduce IO trajectory *io-traj-4* (cf. Figure 6.1), which has its input events occurring at the same time points as *io-traj-2*. This IO trajectory is observed from Repair-Station-2, and thus adheres to a *Priority Ranking (PR)* discipline.

```
io-traj-4:  ((si-0 ()) (in b 1) (out nil 1))
            ((si-1 (b)) (in a 1) (out a 1))
            ((si-1 (b)) (in nil 1) (out b 1))
            ((si-0 ()) (in nil 3) (out nil 3))
            ((si-0 ()) (in a 1) (out a 1))
            ((si-0 ()) (in nil 1) (out nil 1))
            ((si-0 ()) (in c 2) (out nil 2))
            ((si-1 (c)) (in b 2) (out b 2))
            ((si-1 (c)) (in nil 1) (out c 1))
            ((si-0 ()) (in nil 1) (out nil 1)).
```

The policy adhered to by Repair-Station-2 entails more than just a regular priority ranking of the waiting queue. Newly arriving vessels of type "B" preempt vessels of type "C" that are currently being serviced, and newly arriving vessels of type "A" preempt all vessels of other types.

I shall assume that *io-traj-1* and *io-traj-4* have been actually observed. This leads to the following entries in LTMS:

```
 1. (((SI-1 (C)) (IN NIL 1)) ((SI-0 ()) (OUT C 1))):  TRUE
 2. (((SI-1 (C)) (IN B 2)) ((SI-1 (C)) (OUT B 2))):  TRUE
 3. (((SI-1 (B)) (IN NIL 1)) ((SI-0 ()) (OUT B 1))):  TRUE
 4. (((SI-1 (B)) (IN A 1)) ((SI-1 (B)) (OUT A 1))):  TRUE
 5. (((SI-0 ()) (IN C 2)) ((SI-1 (C)) (OUT NIL 2))):  TRUE
 6. (((SI-0 ()) (IN NIL 3)) ((SI-0 ()) (OUT NIL 3))):  TRUE
 7. (((SI-0 ()) (IN B 1)) ((SI-1 (B)) (OUT NIL 1))):  TRUE
 8. (((SI-0 ()) (IN C 3)) ((SI-0 ()) (OUT C 3))):  TRUE
 9. (((SI-0 ()) (IN B 2)) ((SI-0 ()) (OUT B 2))):  TRUE
10. (((SI-0 ()) (IN NIL 1)) ((SI-0 ()) (OUT NIL 1))):  TRUE
11. (((SI-0 ()) (IN A 1)) ((SI-0 ()) (OUT A 1))):  TRUE
```

It can be noticed at once that the created database above contradicts that obtained for the FIFO discipline. In particular, the two entries #4

```
4.  (((SI-1 (B)) (IN A 1)) ((SI-1 (A)) (OUT B 1)))  FIFO
4.  (((SI-1 (B)) (IN A 1)) ((SI-1 (B)) (OUT A 1)))    PR
```

are in conflict with each other. This is an interesting observation. It shows that LDIR is able to distinguish between the two disciplines on the basis of just a few observations alone.

Similar to the earlier scenarios, several prospective output trajectories can be predicted for the previously used input trajectory *in-traj-3* in accordance with the selected assumption sets. Since we understand the internal working of Repair-Station-2, we can, of course, know the true behavior of the system as it deals with *in-traj-3*. For Repair-Station-2 (RS-2), the correct observed IO trajectory *io-traj-3-pr* is:

```
io-traj-3-pr:  ((si-0 ()) (in c 1) (out nil 1))
               ((si-1 (c)) (in a 1) (out a 1))
               ((si-1 (c)) (in nil 2) (out c 2))
               ((si-0 ()) (in nil 3) (out nil 3))
               ((si-0 ()) (in b 2) (out b 2))
               ((si-0 ()) (in nil 1) (out nil 1)).
```

In Scenario 4, we shall predict the behavior of RS-2 given the same initial state and assumption sets as in Scenario 1.

```
initial-state:  (si-0 ())
asn-set:  (longest abs-state)
in-traj-3:  (in c 0)
            (in a 1)
            (in b 6)
            (in nil 8)
```

For Scenario 4, the predicted input and output trajectories are (cf. Figure 6.3):

```
Input-Trajectory-IV:  ((SI-0 ()) (IN C 1))
                      ((SI-1 (C)) (IN A 4))
                      ((SI-0 ()) (IN NIL 1))
                      ((SI-0 ()) (IN B 2))

Output-Trajectory-IV:  ((SI-1 (C)) (OUT NIL 1))
                       ((SI-0 ()) (OUT C 4))
                       ((SI-0 ()) (OUT NIL 1))
                       ((SI-0 ()) (OUT B 2)))
```

with the LTMS having added the following 2 hypothesized IO segments:

```
12. (((SI-0 ()) (IN C 1)) ((SI-1 (C)) (OUT NIL 1))):  TRUE
13. (((SI-1 (C)) (IN A 4)) ((SI-0 ()) (OUT C 4))):  TRUE
```

Input abstraction had to be used on the second of the predicted IO segments. For the reduced input segment:

```
((SI-1 (C)) (IN A 4))
```

the following candidate IO segments can be used for abstraction:

```
1. (((SI-1 (C)) (IN NIL 1)) ((SI-0 ()) (OUT C 1))):  TRUE
2. (((SI-1 (C)) (IN B 2)) ((SI-1 (C)) (OUT B 2))):  TRUE
```

out of which segment #1 was chosen, leading to the prediction:

```
(((SI-1 (C)) (IN A 4)) ((SI-0 ()) (OUT C 4)))
```

i.e., RS-2 is also in the ship-sinking business. This time around it was the newly arriving vessel of type "A" that magically disappeared.

A comparison of hypothesized IO segments from the databases corresponding to the similar Scenarios 1 and 4 shows that the hypotheses are also inconsistent with each other. In particular, IO segments #15 from Scenario 1 and #13 of Scenario 4 contradict each other:

```
15. (((SI-1 (C)) (IN A 4)) ((SI-1 (B)) (OUT C 4)))  (FIFO)
13. (((SI-1 (C)) (IN A 4)) ((SI-0 NIL) (OUT C 4)))    (PR)
```

LDIR predicts two distinct IO trajectories `IO-Trajectory-II` and `IO-Trajectory-IV` on the basis of only two IO segments, whereby one of them ({*io-traj-1*}) is identical in the two scenarios, and the other ({*io-traj-2*} vs. {*io-traj-4*}) is influenced by the selected shipyard discipline. Evidently, LDIR is able to distinguish clearly between the two ship servicing disciplines (FIFO vs. PR). Alternatively, it can be said that LDIR will not predict the true IO trajectory *io-traj-3-pr* given the observations *io-traj-1* and *io-traj-2*; similarly, LDIR will not predict the true IO trajectory *io-traj-3-fifo* based on *io-traj-1* and *io-traj-4*. This is encouraging indeed.

As Scenario 5, we shall consider RS-2 with the same initial state and assumption sets that were previously used in Scenario 2:

```
initial-state:  (si-0 ())
asn-set:  (exact abs-input)
in-traj-3:  (in c 0)
            (in a 1)
            (in b 6)
            (in nil 8)
```

Scenario 5 leads to the predicted IO trajectory `IO-Trajectory-V` (cf. Figure 6.3):

```
Input-Trajectory-V:  ((SI-0 NIL) (IN C 1))
                     ((SI-1 (C)) (IN A 2))
                     ((SI-1 (C)) (IN NIL 1))
                     ((SI-0 ()) (IN NIL 1))
                     ((SI-0 ()) (IN NIL 1))
                     ((SI-0 ()) (IN B 2)))

Output-Trajectory-V:  ((SI-1 (C)) (OUT NIL 1))
                      ((SI-1 (C)) (OUT B 2))
                      ((SI-0 ()) (OUT C 1))
                      ((SI-0 ()) (OUT NIL 1))
                      ((SI-0 ()) (OUT NIL 1))
                      ((SI-0 ()) (OUT B 2)))
```

As before, the second IO segment called for input abstraction. The same two IO segments from LTMS were available for abstraction. This time, LDIR chose the other IO segment #2. As mentioned earlier, LDIR picks one arbitrarily in this situation.

It can be noticed that the predicted `IO-Trajectory-V` resembles more closely the observed *io-traj-3-pr* than the previously predicted `IO-Trajectory-IV`. The same could be concluded when comparing the first three scenarios with the observed *io-traj-3-fifo*. It seems that choosing `elm-1: exact` leads to better predictions than `elm-1: longest`. It also seems that choosing `elm-2: abs-input` is superior to `elm-2: abs-state`. In the following section, I shall try to quantify this observation somewhat.

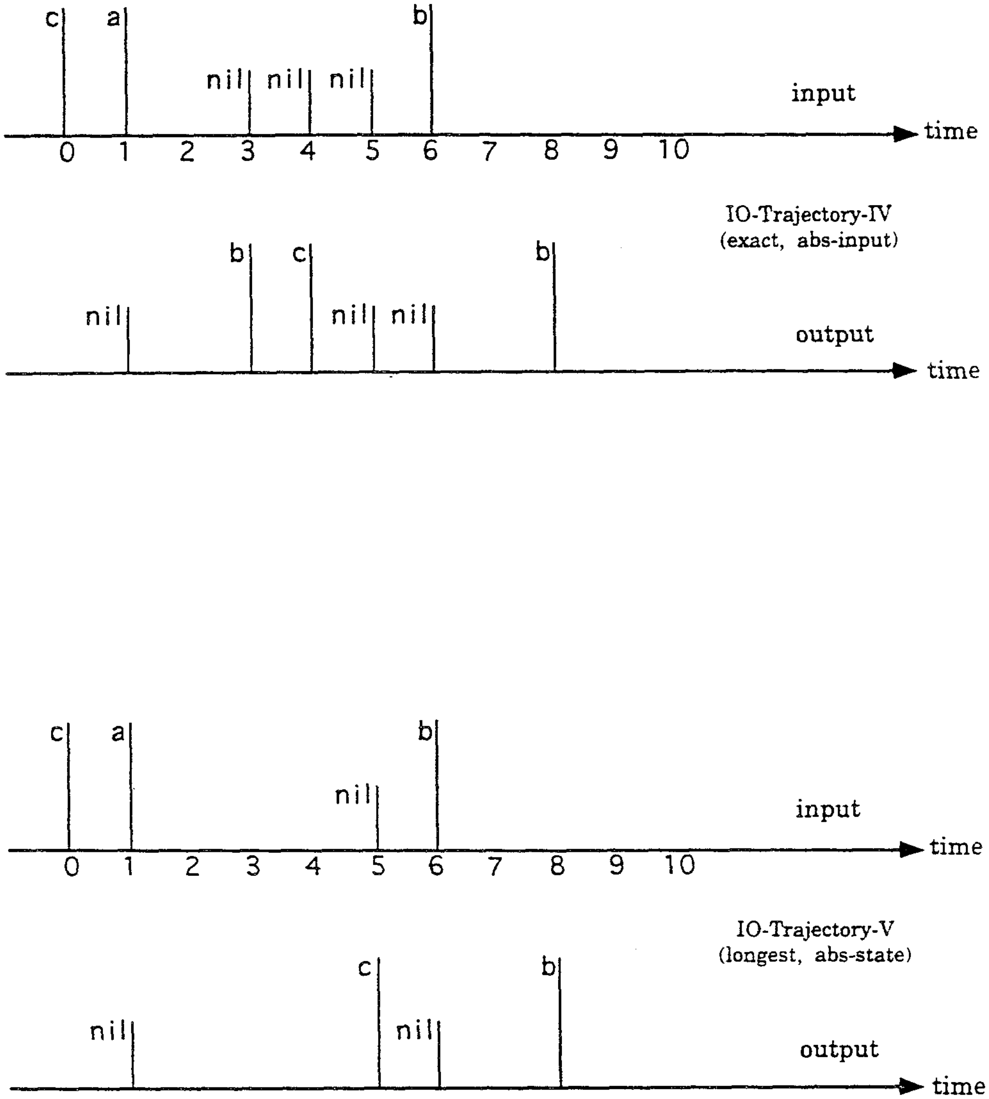
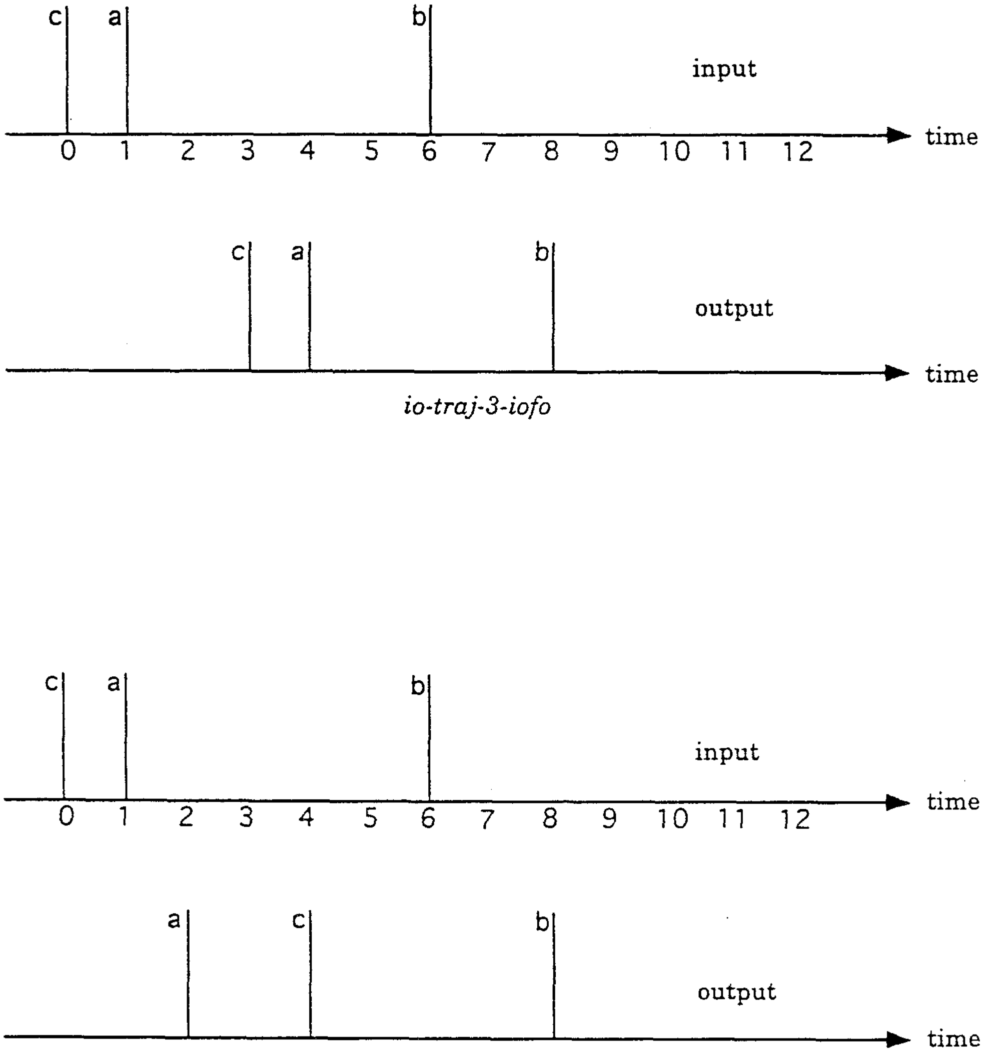


Figure 6.3, Predicted IO trajectories (PR discipline): `IO-Trajectory-IV` and `IO-Trajectory-V`



io-traj-3-iofo

io-traj-3-pr

Figure 6.4, Correct observations for FIFO & PR disciplines given *in-traj-3*.

6.5 Quantitative Evaluation of Predicted IO Trajectories

I have already mentioned that the early chapters of this dissertation present the foundations of the methodology are laid in a rather formalistic fashion, whereas the later chapters turn more and more to the tools of the experimentalist to provide practical meaning and engineering relevance to the formalistic constructs introduced earlier. It was unavoidable that, while turning to more and more applied issues, my terminology also changed somewhat. The early chapters of this dissertation are written in the language of formal logic, whereas this chapter is written using a pseudo-lisp notation to explain the algorithms implemented in the LDIR software.

In Chapter 5, I developed (rather informally) algorithms to hypothesize new IO segments from a (non-empty) set of available IO segments and some assumptions (abstractions). At this point, I wish to provide a definition of how IO segments are generated by DIR in the language used in Chapter 4, in order to bridge the gap between these chapters and help the user to understand how the various terminologies and nomenclature used in this dissertation fit together.

In the literature, the symbol $\mid\!\sim$ is commonly used to denote non-monotonic inference. Since DIR (non-monotonically) hypothesizes unobserved IO segments based on assumptions, I use the symbol $\mid\!\sim_{abs}$ to emphasize the use of abstractions. An IO segment α is said to be *non-monotonically derivable* from an input segment α_{in} and a non-empty set of available IO segments Δ under a given set of assumptions, i.e., $\Delta \cup \{\alpha_{in}\} \mid\!\sim_{abs} \alpha$, if it can be hypothesized from the available IO segments using the abstraction mechanisms specified in the assumption set. In this way, we can refer to non-monotonic prediction of IO segments in a somewhat well-defined context.

I now wish to assess the quality of the database maintained by LTMS, and the effectiveness of LDIR in using the available knowledge for predicting IO trajectories. We can say that the database maintained by LTMS constitutes the *model* of the system, whereas predicting an IO trajectory represents a qualitative *simulation.* Thus, assessing the quality of the database is synonymous with *validating the model*, whereas assessing the quality of a prediction corresponds to *verifying the simulation.*

More traditional modeling/simulation systems use goodness-of-fit measures to assess the quality of a simulation. Also, since the model usually remains static during the simulation, it is possible to separate the task of validating the model from that of verifying the simulation. In LDIR, and in other qualitative modeling/simulation systems, these tasks are more intricate. In LDIR, the model does not remain static during the simulation (new hypotheses are added on the fly, and the available knowledge is constantly revised to maintain consistency among all available facts and hypotheses). Also, the modeling and simulation engines are basically the same algorithms applied in slightly different fashions.

Before I can define precisely what I mean by the quality of the database (the model) in the context of LDIR, we need to understand what the term "quality" entails. The quality of a knowledge maintenance system always has two components that must be assessed separately, and that are always in competition with each other. I shall illustrate these two properties with a simple example.

The warehouse of a large company stores many parts that it must sell when they are needed. The warehouse manager keeps a book in which all parts are listed with their part numbers, location in the warehouse, and prices. Unfortunately, the knowledge is incomplete. There are many parts for which no prices are listed, and other parts are missing altogether. The warehouse manager now has two choices: he can work with the book as is. In this case, the *reliability* of the available knowledge is excellent, yet the *completeness* is not. If the apprentice is to sell a part for which no price is listed, he must always look for the warehouse manager to find out how much to charge. The other possibility is for the apprentice to estimate the cost of the incomplete entries as best as the manager can within the limitations of available time. As missing entries are added to the book, the better will the knowledge becomes more *complete*. Unfortunately, the *reliability* will suffer in the process.

The same is true for the database maintained by the LTMS. The more hypotheses are added to the database, the more complete the knowledge becomes. However, the increase in completeness goes hand-in-hand with a reduction in reliability. Thus, when assessing the quality of the database, we shall define two separate quality measures — one assessing the reliability of the database, and the other assessing its completeness.

How do we assess completeness? I begin with a number of (fairly informal) observations about the *cardinality* of the spaces of hypothesized IO segments. Given a set of input segments, let their space of hypothesized IO segments refer to those hypotheses that are non-monotonically derivable based on some observed IO segments and a unique abstraction type. For example, using abstraction `abs-length` with some observed IO segments leads to a space of hypothesized IO segments for a set of input segments. Another space of hypothesized IO segments for the same set of input segments and observed IO segments can be non-monotonically derived using `(abs-length, abs-input)`. Each of the three abstraction types (`abs-length`, `abs-input`, and `abs-state`), usually leads to different hypothesized IO segments (cf. Section 6.3). Of course, it is not necessary that the IO segments from these three spaces are mutually exclusive. The completeness measure must somehow be related to the cardinality of observed, hypothesized, and hypothesizable IO segments. A more precise definition will be given later.

How can reliability be assessed? Each abstraction type enforces a certain degree of *predictability.* We can rank the three types of abstraction in order of increasing *compliance* as: (1) length, (2) input event, and (3) state. I am using the term "compliance" to indicate how forgiving an abstraction is (similar to the compliance of a spring). The more compliant an abstraction is, the less reliable a hypothesis will be, and the smaller the fidelity of a predicted IO segment that is based on this hypothesis. State abstractions make much stronger assumptions in generating hypothesized IO segments than either input event or length abstractions. I shall illustrate this fact in due course with our shipyard example. Of course, it is also possible to allow combinations of abstractions, as in the sample scenarios. Having the ordering defined as given above, it is fairly straightforward to order their combinations as well. If length, input event, and state are abstracted simultaneously, then the so hypothesized IO segments are based on the most compliant assumptions, allowing prediction of *anything* as long as consistency is not violated!

The compliance of an abstraction type offsets the *fidelity* of IO segments hypothesized based on it. In the most restrictive sense, the term "fidelity" might refer to whether a hypothesized IO segment will eventually be confirmed or rejected. However, instead of postulating a dichotomy, I use the term fidelity to indicate to what extent a hypothesized IO segment is consistent with the available IO segments. The stronger the compliance of an abstraction, the worse the fidelity of hypothesized IO segments generated based on it — the fidelity of IO segments degrades as abstractions with higher compliance are used. Thus, I use the terms compliance and fidelity in relation to each other.

How does the predictability of LDIR relate to the number of observed IO segments? The answer to this question depends on the candidate input trajectory. The more IO segments have been observed for a system in relation to a candidate

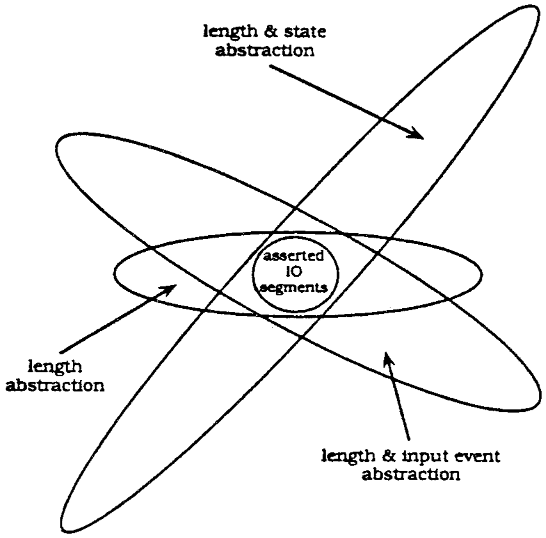


Figure 6.5, Spaces of hypothesized IO segments generated based on some asserted IO segments and different types of abstractions.

input trajectory, the fewer IO segments would need to be hypothesized. Also, more observed IO segments place more restrictions on what IO segments can be hypothesized.

Hence the size of the space of the hypothesized IO segments is related to both the number of observed IO segments and the types of abstractions used. In general, the cardinality of the space of hypothesizable IO segments based on `(abs-length, abs-state)`, `(abs-length, abs-input)`, or `(abs-length)` decreases in the order given. Furthermore, while the number of hypothesizable IO segments decreases as less compliant types of abstractions are employed, their fidelity improves. Figure 6.5 shows the cardinality of spaces for hypothesized IO segments corresponding to three abstraction types in relation to a set of asserted IO segments.

Having different types of IO segments (some asserted, others hypothesized), it is possible to define two heuristic quality measures relating to: (1) the quality of

ASSUMPTION USED	FOM
—	4.0
`exact`	3.5
`longest`	3.0
`(exact, abs-input)`	2.5
`(longest, abs-input)`	2.0
`(exact, abs-state)`	1.5
`(longest, abs-state)`	1.0
`(exact, abs-input, abs-state)`	0.5
`(longest, abs-input, abs-state)`	0.0

Table 6.1, Figures of merit assigned to some types of assumptions.

all available IO segments in the database, i.e., the quality of the model, and (2) the quality of a predicted IO trajectory given an input trajectory, i.e., the quality of a simulation. A *quality measure* is a real-valued number in the range 0.0 to 1.0, where larger values denote improved quality [Cel91].

Now let me introduce the term *IO segment fidelity* in terms of a *Figure of Merit (FoM)* assigned to it. Let F_{abs_i} denote the absolute fidelity of the i^{th} IO segment. Asserted IO segments have the highest figure of merit, whereas hypothesized IO segments using the abstraction mechanism with the largest compliance have the lowest figure of merit. I only consider the possibilities from the first two elements of `elm-1` $\in$ {`longest, exact, all`} together with `elm-2` $\in$ {`abs-input, abs-state`} with some (arbitrarily) assigned figures of merit for each abstraction (assumption) type. Table 6.1 suggests some assigned FoM for each combination of abstraction types. In the chosen scale, the maximum and minimum absolute fidelity values are $F_{\mathrm{max}} = 4.0$ and $F_{\mathrm{min}} = 0.0$, respectively.

Now, it is easy to define the *relative fidelity* of each IO segment. Let:

$$F_{\mathrm{rel}_i} = \frac{F_{\mathrm{abs}_i}}{F_{\mathrm{max}}}$$

denote the relative fidelity of an IO segment. Assertions have the highest relative fidelity (1.0), whereas hypotheses that are based on the abstractions with the highest compliance have the lowest relative fidelity (0.0). Evidently, the relative fidelity can be used as a quality measure.

Given all currently available IO segments in the LTMS database, we can define the *average relative fidelity* as:

$$F_{\mathrm{avg}} = \frac{1}{n} \cdot \sum_{i=1}^{n} F_{\mathrm{rel}_i}$$

where n denotes the number of IO segments currently stored in the database. F_{avg} can also be used as a quality measure. We shall use F_{avg} as the *reliability measure* of our model.

Given that we have a finite set of Complete input segments, each either being a hypothesis or an assertion, let us call the number of all possible combinations of initial states and input events N_p[3]. Also, let us call the set of all combinations of different initial states and input events that are currently present among the (asserted and hypothesized) IO segments N_a, where $N_a \leq N_p$.

We can introduce the *evidence ratio measure*:

$$E_R = \frac{N_a}{N_p}, \qquad 0.0 < E_R \leq 1.0$$

Evidently, E_R can also be used as a quality measure. We shall use this measure as the *completeness measure* of our model. The interpretation of E_R is that, the

[3] Often, the number of possible states the system can be in is infinite. I shall discuss in due course how we handle this situation.

greater the value of E_R, the fewer IO segments will need to be hypothesized in relation to a fixed set of initial states and input events. Smaller values of E_R indicate the converse.

A value of $E_R = 1.0$ indicates that LDIR will never have to resort to either input or state abstractions in hypothesizing IO segments. This measure ignores length abstractions, as they are in most cases fairly harmless.

Now, we can use F_{avg} and E_R to define Q_{Model}, the *prediction quality* of the model, i.e., the overall quality of all available IO segments in LTMS:

$$Q_{\text{Model}} = F_{\text{avg}} \cdot E_R.$$

The two influencing factors of the prediction quality Q_{Model} are always in competition with each other. Evidently, if the cardinality of all possible IO segments is exhausted through observations, $Q_{\text{Model}} = 1.0$. A useful feature of quality measures, as they were defined in [Cel91], is that multiple (usually competing) quality measures can be simply multiplied with each other, leading to a multidimensional new quality measure that takes into consideration all the influencing factors assessed through the individual quality measures contributing to it.

Evidently, if no hypotheses have been made, F_{avg} shows a perfect score of 1.0. However, when only few IO segments are observed or hypothesized, E_R will be poor. On the other hand, if everything has been hypothesized that can be, E_R shows a perfect score of 1.0, but this time around, F_{avg} will be poor.

The *average fidelity measure*, F_{avg}, and the *evidence ratio measure*, E_R, are adaptations of the *entropy reduction measure*, H_R, and the *observation ratio measure*, O_R, used in SAPS [Cel91]. SAPS defines the quality of its inductive model in a similar fashion as:

$$Q_M = H_R \cdot O_R$$

It uses Q_M to distinguish between the relative virtues of different "masks" (the abstraction mechanism used in SAPS), and chooses as the "optimal mask" the one with the largest Q_M value, i.e., it selects the abstraction mechanism that maximizes the predictability power of the qualitative model. LDIR currently does not need to do so, because there are a small number of choices available in selecting abstraction mechanisms, and their relative merits w.r.t. predictability power are fairly well understood. In the future, however LDIR will be expanded to deal with multi-input systems (as SAPS already does); at such time, Q_{Model} may be used by LDIR as Q_M is currently used in SAPS.

If we don't know yet what we wish to use the model for, i.e., which input trajectory we are going to use to predict an output trajectory, this is the best we can do. However, given an input trajectory, we can evaluate the quality of the IO trajectory predicted by LDIR in more direct ways,i.e., we can predict the *quality of a simulation.*

Since we already know the input segments we shall have to work with, the evidence ratio is of no concern any longer. We only deal with the relative fidelities of individual predicted IO segments. Making the supposition of statistical independence of neighboring predicted IO segments, we can postulate the following quality measure:

$$Q_{\text{Simul}} = \prod_{j=1}^{m} F_{\text{rel}_j}$$

where m denotes the number of predicted IO segments. The supposition of statistical independence is obviously a preposterous one. Simulation output is *never* statistically independent (unless we try to predict the next value of a noise generator). However, we don't have anything better to go by, and so we shall have to live with this supposition. Most qualitative simulation systems do. For example, SAPS does exactly the same, except that F_{rel_j} is replaced by a *measure of likelihood* of the prediction made.

6.5.1 Evaluation of the Shipyard Example

Let us now use the measures defined in the previous section to analyze the predicted Scenarios 1 through 5 of the shipyard example.

When assessing the evidence ratio for the shipyard example, we are immediately faced with a problem: the number of different possible states is infinite, since an arbitrary number of ships can be waiting in the input queue. Consequently, we need to replace the theoretical cardinality by a much smaller subset: the power set of all initial states and input events that have ever been observed.

Unfortunately (or fortunately), we can't know what we don't know. We have to live with this fact, and make the best of it. Luckily, the powerset of all ever observed states is something that LDIR can compute. In the case of RS-1, LTMS contains 4 different initial states and 4 input event types. The power set of these two (independent) quantities is 16. In the case of RS-2, only 3 initial states have ever been observed, since the state `(SI-1 (A))` has never been seen by the system. Evidently, no abstraction mechanism in the world will enable LDIR to make a prediction of this state ... until the state shows up for the first time in an observation. Thereafter, we are dealing with an entirely new situation, and the evidence measure will have to be revised.

Before proceeding with the remainder of this section, I note that the tables given here are based on few data points. Thus, not much emphasis should put on them!

Table 6.2 shows the fidelity measure, F_{avg}, and the evidence measure, E_R, of

Observed IO segments	F_{avg}	$E_R = Q_{Model}$
Repair-Station-1 (FIFO)	1.0	0.5
Repair-Station-2 (PR)	1.0	0.6667

Table 6.2, Quality measures for observed IO segments from repair stations 1 & 2.

the databases of the two repair stations, RS-1 and RS-2, before any additional IO segments were hypothesized.

Evidently, the fidelity measure shows a perfect score, since the databases don't yet contain any hypotheses. Given the observed IO segments only, the prediction quality of IO segments for the RS-2 is evidently better than that of RS-1. This is because LDIR is more ignorant (innocent) in the case of RS-2, since it has never seen the state `(SI-1 (A))`.

Given that there are 4 distinct initial states and 4 input events for RS-1, the total number of combinations of initial states and input segments is 16. The observed IO segments contain 8 different cases. Therefore, $E_R = 8/16$. Considering RS-2, the number of distinct initial states is 3; thus, LDIR can only imagine 12 combinations. The observed IO segments for RS-2 also contain 8 different cases, which leads to $E_R = 8/12$.

Now, we can look at quality measures for Scenarios 1 through 3 (cf. Table 6.4), and compare them with each other. The quality measures of the model change since the model itself is updated to include some hypotheses.

Looking at the quality of the model, we may conclude that this model is best suited for predictions. However, the quality of the one predicted trajectory using this model is dismal. Evidently, the fact that 4 of the 15 IO segments in the database are hypotheses, has not been punished enough. This can easily be done by proposing a third quality measure, the *assertion ration measure*:

	F_{avg}	E_R	Q_{Model}	Q_{Simul}

`IO-Trajectory-I`
`(longest, abs-input)`
	0.9	0.625	0.5625	0.1406

`IO-Trajectory-II`
`(exact, abs-input)`
	0.9615	0.5625	0.5409	0.5469

`IO-Trajectory-III`
`(longest, abs-state)`
	0.9423	0.5625	0.53	0.3282

Table 6.3, Quality measures for Scenarios 1 through 3 (Repair-Station 1).

	F_{avg}	E_R	A_R	Q_{Model}	Q_{Simul}

`IO-Trajectory-I`
`(longest, abs-input)`
	0.9	0.625	0.7333	0.4125	0.1406

`IO-Trajectory-II`
`(exact, abs-input)`
	0.9615	0.5625	0.8462	0.4577	0.5469

`IO-Trajectory-III`
`(longest, abs-state)`
	0.9423	0.5625	0.8462	0.4485	0.3282

Table 6.4, Quality measures for Scenarios 1 through 3 (Repair-Station 1).

$$A_R = \frac{N_A}{N_A + N_H}$$

where N_A denotes the number of assertions in the database, and N_H represents the number of hypotheses. The modified model quality would then be evaluated as:

$$Q_{\text{Model}} = F_{\text{avg}} \cdot E_R \cdot A_R$$

Using the modified model quality measure, we obtain the following modified table:

Now, there is a good correspondence between what the model quality stipulates and what the simulation quality confirms. The original definition is left in the text to show that there is a fairly high degree of heuristicism in the detailed definitions

	F_{avg}	E_R	A_R	Q_{Model}	Q_{Simul}
IO-Trajectory-IV (longest, abs-input)	0.9423	0.75	0.8462	0.5980	0.375
IO-Trajectory-V (exact, abs-input)	0.9615	0.75	0.8462	0.6102	0.5469

Table 6.5, Quality measures for Scenarios 4 and 5 (Repair-Station 2).

of these quality measures, and more fine tuning may be needed down the road; for now, however the modified quality measures look rather promising.

Note that the evidence ratio has gone up in all cases (as it must), yet the fidelity measure and the assertion ratio have both gone down, and the overall model quality has in fact decreased, i.e., we didn't do a very smart thing by augmenting the LTMS with these hypotheses. In all likelihood, at least some of them will have to be revoked (negated) in the future. Of course, we already know this to be true from Section 6.3.3.

Among the three cases, both the model quality measure and the simulation quality measure suggest that Scenario 2 is the best. Scenario 1 added so many spooky hypotheses to the LTMS that its results are the most doubtful ones, although the abstraction mechanism is valued more reliably, in general, than that used by Scenario 3. The model and simulation quality measures agree in their relative assessments of the three scenarios.

Let me now discuss the case of Repair-Station 2, i.e., Scenarios 4 and 5. The tabulated results are as follows:

This time, the assessments of the model quality come in higher. The reason is that (as far as LDIR knows), this system has a smaller number of degrees of freedom (a smaller cardinality); thus, there is less uncertainty when making guesses. Again, the model and simulation quality measures are in good agreement with each other.

Now, I examine each of the predicted scenarios in terms of their behaviors. I begin with IO-Trajectory-I. Evidently, the predicted IO trajectory is not the expected one. Moreover, the order in which vessels are predicted to be repaired is incorrect. Even though the first predicted IO segment is consistent and correct, the remaining part of the predicted IO trajectory is incorrect. LDIR predicted vessel "C" to take longer than expected to be repaired. Consequently, all the remaining IO segments turn out to be incorrect. This, obviously, need not be true in general. The reason vessel "C' requires 5 time units for repair is due to asking for the longest match in the assumption set. Nevertheless, the number of vessels entering and leaving the repair-station-1 is correct.

How about IO-Trajectory-II? The predicted IO trajectory satisfies the FIFO discipline. This is accidental however, since inside the shipyard two major "remodeling jobs" have taken place that were not visible from the outside. The amount of time predicted for vessels "C" and "A" to be serviced is incorrect. Moreover, IO segment (((SI-0 ()) (IN NIL 2)) ((SI-0 ()) (OUT NIL 2))) is split into two identical IO segments (((SI-0 ()) (IN NIL 1)) ((SI-0 ()) (OUT NIL 1))). This is due to asking for exact match and not having (((SI-0 ()) (IN NIL 2)) ((SI-0 ()) (OUT NIL 2))) in IO-space-g. Evidently, this deviation is harmless. IO-Trajectory-II looks better than IO-Trajectory-I, but the conclusion may be accidental, since the internally made abstractions are still rather dubious.

(IO-Trajectory-III) shows the least agreement with the desired predicted IO trajectory. We expected this since the assumption set is (longest, abs-state). The abstraction of initial states is more compliant than that of input events. In this scenario, vessel "C" disappeared altogether!

The above discussion shows that assessing a qualitative simulation in the same way as one would judge a quantitative simulation, i.e., in terms of a goodness-of-fit measure, is problematic at best. The previously presented quality measures are much more solid and reliable, in general, than any goodness-of-fit measure we might come up with.

If this is the case, what is the purpose of the simulation? Had we provided the system with more evidence (a higher evidence ratio) to start with, the quantitative results of the simulation would also have been better. The problem is simply that if the agreement between reality and prediction is poor, as in the shipyard example, this is related to the problem of not having enough evidence, and not to a principal flaw in the methodology. The proposed quality measures have a much better chance of assessing the real strengths and weaknesses of such a model than a simple output-to-output comparison.

However, there is also another implicit benefit of performing logic-based qualitative simulation runs. It relates to the possibility of understanding the underlying reasoning processes of the qualitative simulator. A traditional weakness of simulation is that simulation results are rarely enlightening [Cel91]. It is as difficult to generalize knowledge from a simulation output as from a lab experiment. Many different simulation runs are usually needed until a human researcher can discern the general patterns behind the specific patterns generated by individual simulation runs. In this respect, the DIR methodology exhibits an important advantage. Its reasoning processes are immediately open to human interpretation. I shall talk more about this facet of the DIR methodology in the following section.

6.6 Evaluation of DIR

Before evaluating DIR, it is helpful to discuss the basic characteristics of inductive modeling/reasoning in general settings. An inductive modeler/reasoner (e.g., [CM83, CM86, GN87, BZ87, MK83, Cel91, Gin93]) is basically comprised of a *modeling engine* and a *simulation engine.* The former creates a model from some finite dataset by resorting to abstraction. The latter uses the model to make predictions (cf. Figure 6.6). Usually, there exists a strict separation between the modeling and simulation engines. Whereas the process of model creation is frequently *inductive*, the once derived model seems to be generally used in a *deductive* fashion during the process of simulation[4].

The goal in creating the model is to allow exactly one prediction in every situation. In other words, the ideal model would be deterministic. However, due to lack of data, often such an ideal model cannot be created without abstracting too much. This is due to the fact that, at a desirable granularity level, data may either support multiple predictions or none at all [Cel91]. To resolve the problem of multiple predictions, one or more of the following strategies can be used: (1) collect additional data, (2) add more constraints or make the existing constraints more stringent, (3) use figures of merit to choose among the alternatives, and (4) consider all possibilities[5]. In the case that no prediction is possible, fewer constraints or less restrictive constraints, and more compliant abstractions can be useful.

What about the simulation engine? What does it do? In its most basic form, it is a pattern matcher. For any input that it receives, it finds its corresponding output based on the model it operates on. However, it may take an active role in dealing with the situation when either multiple or no predictions are possible, yet without changing the model that it uses. In a more synergistic manner, it is also possible for the modeling and simulation engines to interact with each other in order to improve the model and overcome some of its restrictions (e.g., multiple predictions). The discovery of no possible predictions is the responsibility of the

[4] Deductive models have a much higher validity value if they can be devised. This may explain why existing inductive modeling methodologies are so keen on obtaining deductive models from observed data.

[5] This is what is usually done in deductive qualitative modeling. This approach usually leads to excessive branching, and is therefore only applicable when dealing with simple systems.

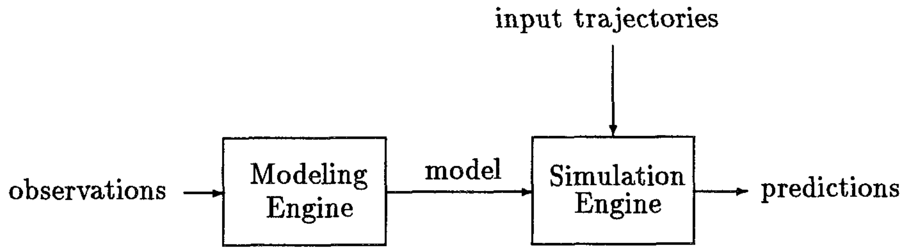


Figure 6.6, Making predictions from observations.

simulation engine.

How does DIR relate to Figure 6.6? We know that DIR also operates on a model, namely an iterative IOFO specification. However, the IOFO specification is not a static model. Both newly arriving observations and previously made hypotheses constantly change the contents of the databases, and thereby the model. Hence, while (other quantitative and qualitative) simulation systems that operate on a fixed model in a purely deductive fashion, DIR also induces new knowledge during the simulation phase.

The iterative IOFO model consists of nothing more than a set of observed IO segments with their associated states. Whereas the presence of states is a mild indication of structure, the Complete IO segments are disjointed from each other and do not constitute a structural model. An iterative IOFO does not specify any explicit structure among IO segments.

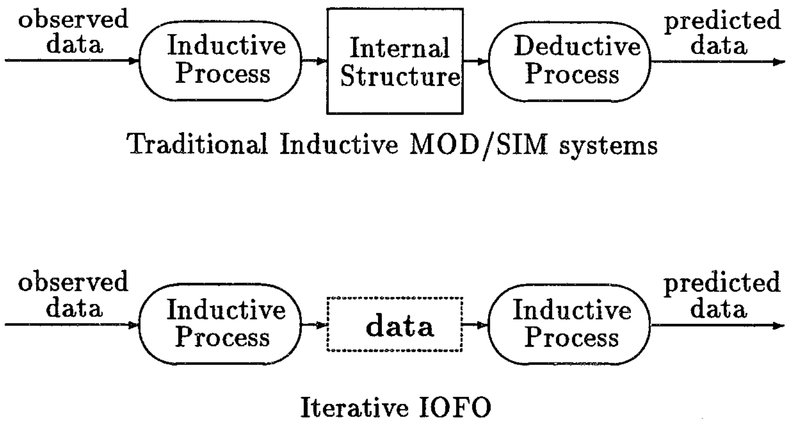
Figure 6.7 shows the comparison of DIR with more traditional inductive modeling/simulation systems. Most inductive modeling tools make assumptions about the model structure and then use parameter estimation techniques minimizing the distance between observed and simulated behavior. Once this process is concluded, the model is in no way different from a deductively obtained structural model, and the same simulation engine can be used for both.

Other techniques, such as the well-known neural network approach [Gro88], relax the requirement of prespecifying a structure somewhat, in that they operate on a more general structure that can be fitted to various types of behavior. However, neural network models are also parametric in nature. Different neural networks may use different internal structures, and they may vary in the training algorithms used for parameter optimization; however, once the network has been trained, it can again be simulated using standard simulation engines. Some neural networks don't operate on a fixed model in that they keep learning while they simulate.

Other techniques, such as the System Approach Problem Solver (SAPS) [Cel91], are non-parametric, just like DIR. Also, SAPS operates simply on a bunch of unconnected data records, trying to find an optimal match between the current input pattern and previously observed similar input patterns, using an interpolated value of their outputs as the predicted new output value. However, contrary to DIR, SAPS strictly separates the simulation task from the modeling task, and, during simulation, the model remains fixed.

Most AI-based inductive systems [CM83, CM86, MK83, GN87] use non-temporal logic-based languages such as VL_{21} [LM77]. Consequently, they cannot easily represent time-dependent facts or beliefs. Of course, non-temporal logic can emulate temporal logic. SAPS, for example, does this by treating past input and output values as additional "current" inputs. To do so just requires additional work for encoding the knowledge and providing sound semantics for it. The reasoning part of DIR also uses non-temporal logic[6]; consequently, it is unable to reason about the time content of its knowledge — there is none. However, the inference engine of DIR is making use of time explicitly. It does not use any form of theorem prover, be it a temporal or a non-temporal one; the implication of using time explicitly

[6] Recall that an IO segment was abstracted to a proposition (cf. Section 5.4.1).



Traditional Inductive MOD/SIM systems

Iterative IOFO

Figure 6.7, A block diagram showing steps in predicting IO trajectories using deductive and inductive modeling approaches.

in making decisions allows DIR's inference engine to account for causality among what it knows about its assertions and hypotheses. As we already know, an output segment is supposed to be *caused* by an input segment. Furthermore, the output state of one IO segment must be identical to the input state of the subsequent one. In this respect, DIR appears to be better equipped than other logic-based systems to deal with situations where it is necessary to use causality explicitly in making hypotheses while maintaining consistency among all available facts and beliefs.

Among the existing inductive modeling approaches, the inductive modeling approach conceptualized by [Kli85] and subsequently advocated and extended by [Cel91] is the one closest to DIR. This methodology is commonly referred to as the *General System Problem Solving (GSPS)* approach to inductive modeling, and the specific implementation discussed in [Cel91] is called the *Systems Approach Problem Solver (SAPS)*. In the sequel, I shall give a brief overview of what SAPS does, and how this relates to DIR. For a comprehensive exposition of GSPS and SAPS, I refer the reader to [Kli85, Cel91].

SAPS is an inductive modeling approach that also operates on observed input/output trajectories. SAPS was designed to inductively model the behavior of continuous-time Multi-Input Single-Output (MISO) systems. It provides guidelines converting continuous trajectories into their counterpart discrete episode. The conversion process is called *recoding*. The dynamics of the system are captured by considering not only the current input values as system inputs, but also past values of all the inputs and of the output.

SAPS uses the concept of a *mask* to select from among all the possible inputs (usually many) the subset that is most representative for capturing the input/output behavior of the system. If the subset is too large, often no matching input pattern can be found; if it is too small, we get multiple matchings. Thus, the mask represents an abstraction mechanism. "Modeling" in SAPS simply refers to the process of finding the optimal mask, i.e., selecting the most appropriate abstraction mechanism among many. In this respect, SAPS differs substantially from LDIR. In LDIR, the abstraction mechanism is user-specified, and "modeling" refers to the process of applying that abstraction mechanism to a set of training IO trajectories, cutting them into appropriate IO segments while enlisting the help of LTMS for preserving consistency among them.

SAPS uses quality measures (a reliability measure and a completeness measure) to select the optimal mask among the many candidate masks. Once the optimal mask has been determined, the process of generating the database to be used during simulation is much simpler in SAPS than its counterpart in LDIR. SAPS doesn't care about consistency at all. Its database is full of contradictions. However, multiple occurrences of the same input pattern are stored in the database independently, and a "voting strategy" is used to decide which output is the most likely one. For this reason, it is important that each input pattern be recorded at least five times whenever possible [Cel91]. An advantage of this approach is that it provides the simulation engine with a *measure of likelihood of correctness* of the prediction made, a feature that is currently not provided by LDIR.

Also the process of qualitative simulation is much less sophisticated in SAPS than in LDIR. SAPS simply looks for similar input patterns in its (usually large) database, picks out the five nearest neighbors, and generates as the prediction of the new output a weighted average of the outputs of these five neighbors.

Now, I summarize the major differences and similarities between the two approaches: (1) SAPS is much more sophisticated than LDIR w.r.t. the selection of the abstraction mechanism. In SAPS, this process is fully automated, whereas in LDIR, the abstraction mechanism is user-specified. (2) LDIR is much more sophisticated than SAPS in its reasoning mechanisms. SAPS reasons in a strictly monotonic fashion. Once SAPS convinces itself that the earth carries a methane atmosphere, nothing in the world will ever make it change its mind! (3) LDIR is, due to its non-monotonic reasoning capabilities, much better geared to deal with incomplete information than SAPS is. Because of its "linear mindset," SAPS is stuck if it ever, during simulation, encounters a situation that it has never seen before. SAPS solves this problem by reasoning with several suboptimal masks (several databases) in parallel. In the very worst case, it can still toss a coin. However, by the time SAPS simulates, it no longer knows how to relax its constraints or work with more compliant abstractions. (4) SAPS was designed to deal with continuously changing phenomena. LDIR was designed to deal with discrete-event systems. However, since SAPS discretizes the continuous-time trajectories into discrete-time episodes — which, in turn, can be represented using discrete events — LDIR can also conceptually handle the types of systems that SAPS is geared to work with. The converse is not necessarily true. (5) SAPS is much further developed than LDIR. SAPS is a fully operational software, whereas LDIR is currently only a prototype. (6) SAPS is currently able to deal with much more complex (multi-input) systems than LDIR is. For this reason, it is too early to provide a comparison of the computational efficiency of the two reasoners.

I continue with the evaluation of LDIR's performance. In the literature, the performance of deductive as well as inductive models is generally judged using some kind of goodness-of-fit measure, looking solely at the accuracy of the predictions made by the reasoner. In this way, the evaluation criteria only implicitly take into account the processes of model generation and simulation. As an example, I mention a theory called *Probably Approximately Correct (PAC)* [Val84]. The term PAC says that a system will probably learn rules that are approximately correct. Given some observations (training examples), it uses probability theory to compute to what extent rules learned are correct for some desired accuracy and confidence levels.

I am, however, as interested in the process of inductive modeling itself as I am in the predictions made by the reasoner. Thus, I assess LDIR's performance along three axes — (1) how accurate its predicted IO trajectories are; (2) how flexible it is in making predictions; and (3) to what extent its underlying reasoning is explicit and accessible to the user. The first measure is a quantitative one, whereas the other two are qualitative measures.

Accuracy is a quantitative measure that can be used to rank alternative predictions. *Flexibility* refers to LDIR's ability to make predictions on the basis of a few observed IO segments alone, i.e., its performance when confronted with incomplete knowledge. Flexibility also includes incremental learning, in that LDIR's predictions should improve as more data are accumulated. Finally, LDIR's *accessibility of reasoning* enables the user to trace back and understand why and how a particular IO trajectory has been predicted. Every hypothesized IO segment has a reason associated with it. The system is responsible for maintaining consistency among all IO segments and their associated reasons for believing in them. These reasons are explicitly available to the user.

Looking at the results shown in Sections 6.3 and 6.4, the performance of LDIR in terms of the accuracy of predicted IO segments (first axis) does not seem very impressive yet. However, this is simply due to the very limited data that LTMS had been fed with. It would be very easy to get more accurate results, but it is much more interesting to watch LDIR's behavior while lacking sufficient data to do the job right. Furthermore, as LDIR was described, it does not rely on any domain-specific knowledge such as different types of bias. In logical approaches, for example so-called *theoretical* bias is often used [GN87][7]. While LDIR uses logical bias (restricting the type of IO segments), it does not use any conceptual bias. Of course, there exists no fundamental difficulty in incorporating domain-specific knowledge into the methodology if so desired. Therefore, LDIR's performance should be judged accordingly. Of course, it is straightforward to use the quality measures introduced in Section 6.5 in order to optimize the abstraction mechanism in use. Instead of relying on the user's insight, the system could use the `all` option for length abstraction, and allow both input and state abstraction. It could then use the quality measures to sort through the predictions made, and come up with the one that offers the overall best quality measures.

Now, I look at the next axis of LDIR's evaluation. One aspect of LDIR's flexibility is that it can predict IO trajectories with few data by making proper abstractions. The lack of sufficient data is in fact the earlier mentioned *Qualification Problem*, and abstraction mechanisms can be used to tackle it. Another aspect of LDIR's flexibility is its ability to learn incrementally. LDIR is not restricted to "one-shot" type learning and hypothesizing of IO segments. As was shown in Section 6.3 and 6.4, other observed IO segments can be added to the already observed IO trajectories as long as they are consistent with the previously stored. There

[7] Theoretical bias refers to *conceptual* and *logical* forms of bias. The former restricts the vocabulary from which sentences can be constructed. The latter restricts the types of sentences (e.g., only allowing conjunctive formulas.)

exists a subtle point here, though. While LDIR is able to detect those IO segments that are causally consistent (inconsistent) with the existing IO segments, it cannot do so for those that are not causally related. In other words, it can only detect consistency (or inconsistency) between two IO segments when they are causally related.

Finally, it is important to know how a particular IO trajectory is predicted and what its merits are (third axis). Due to the underlying explicit reasonings provided by LTMS, we know know why and how an IO segment is predicted. Any predicted IO trajectory generated by LDIR is supported with explicit reasons indicating the type of abstractions used. This is an important feature of LDIR. It distinguishes it from other inductive modeling approaches since its internal reasoning mechanisms are similar to those of a human modeler. LDIR makes decisions (hypotheses) that are supported by the available data, yet it is also capable of making assumptions to enable predictions when the available data are insufficient. It is prepared to revise its predictions whenever warranted. In this way, some of the responsibilities of a human modeler can safely be delegated to LDIR.

I conclude this chapter with a general assessment of DIR's virtues. DIR can operate on few data — it makes hypotheses that are supported by the available data, yet is also capable of making assumptions to enable predictions when the available data are insufficient. It is also prepared to revise its predictions whenever this is warranted. Due to these two features, DIR can tackle the Qualification Problem. Also, with respect to Figure 6.6, DIR does not adhere to a complete separation of the modeling and the simulation engines. Finally, DIR leaves a trace of its decisions that are comprehensible and may prove useful to human modelers.

Chapter 7 Conclusions

This chapter begins with a summary of this dissertation and its contributions, continues with a discussion of the related approaches, and concludes with some suggestions about future research directions.

7.1 Contributions

Given the recent advances in knowledge representation and in particular nonmonotonic reasoning, I proposed an inductive modeling methodology to include abstraction as an integral part of modeling. The ability to integrate abstraction as a feature of a modeling framework can support modeling capabilities that have only been within the reach of human modelers.

I have defined and developed some preliminary steps toward the development of this new inductive modeling methodology for discrete-event systems. I used some observed IO trajectories for single input/single output discrete-event dynamic systems as the starting point. The systems are supposed to be causal, deterministic, and time-invariant. Then, I developed a specification of a system at the level of segmented input/output trajectories. In particular, I took the IOFO specification and derived its iterative specification, G_F, by identifying IO segments such that they can be composed to form the originally observed IO trajectories as well as some which are unobserved. The iterative IOFO system specification resulted in a new abstract stratification of models. I showed that system specification at this finer level enables the inducement of some unobserved IO trajectories from the observed IO segments.

To derive an iterative specification, it was necessary to identify IO segments from IO trajectories. Supposing that only IO trajectories with their initial states are observed, then IO trajectories must be partitioned into IO segments. An assumption set-I was proposed and defined as the underlying basis for partitioning and identifying the IO segments. Based on assumption set-I, granularity levels of input/output variables were determined. Also, the assignment of states to IO segments was handled by the assumption set-I.

I also proposed assumption set-II to deal with the qualification problem. Although the iterative IOFO specification by itself allows some degree of predictability, the assumption set-II is the basis for abstraction. In particular, assumption set-II allows extending the space of predictable IO trajectories by properly using abstractions to deal with lack of complete data.

Based on the iterative system specification, four types of discrete-event input segments were defined. Given these input segments, I derived four types of discrete-event input/output segments. From these segments, two input segments and two input/output segments were selected. Then, another specification of the iterative specification, $\widehat{G_F}$, was formulated containing these well-defined IO segments with their initial and final states being either observed or predicted. This new formulation of the iterative specification was based on the unique representation of IO segment pair. I continued with defining three types of equivalence relations (i.e., length-equivalence, input-equivalence, and state-equivalence) for discrete-event segments. These equivalence relations provided the basis for abstracting length, input-event, and/or state of an IO segment to construct a new IO segment from it.

To handle assumption set-II appropriately and consistently, I argued that traditional reasoning strategies are inappropriate. Then, I proposed using non-monotonic reasoning and in particular a Logic-based TMS. I discussed how inductive modeling is a new application area of NMR. Using the system specification $\widehat{G_F}$, the equivalence relations, and the LTMS, I arrived at what I had hoped for — the skeleton of a Discrete-event Inductive Reasoner. The LTMS's mechanisms were shown to be appropriate for handling the non-monotonicity inherent in inductive modeling.

Since DIR is essentially a problem solver, it was necessary to make some choices about how to represent IO segments of the iterative specification for use in the LTMS. Since we are interested in simple forms of abstractions, I chose to represent IO segments within the language of Propositional Calculus.

I showed how equivalence relations can be used in conjunction with LTMS to construct and eventually hypothesize IO segments based on length, input event, or/and state abstractions. A precise vocabulary was developed to distinguish IO segments with or without assigned beliefs. The existing IO segments with assigned beliefs were classified as assertions or hypotheses. Likewise, the candidate (observed or constructed) IO segments without any belief assignment were classified as observed or predicted. Then, I developed a set of consistency axioms to aid DIR in maintaining consistent IO segments. When these axioms were satisfied, addition of the candidate IO segments to the existing IO segments was granted since consistency was assured. The faith of candidate IO segments was tabulated w.r.t. the consistency axioms and the existing IO segments.

I showed how to partition an input trajectory into candidate input segments in order to predict consistent IO trajectories. To determine a candidate input segment from an input trajectory, I defined three choices. Two of the choices indicate whether predicted IO trajectories should be constructed using IO segments with maximum length or with exactly the same length as those already observed or hypothesized. The third choice allows prediction of IO trajectories that are constructed from IO segments with maximum length, exact length, or any length in between. With the ability to partition an input trajectory and predicting IO segments based on the available data and assumption set-II, DIR has the means to predict IO trajectories.

I have implemented a prototype of DIR in Common-Lisp. The implementation, called LDIR, consisted of the design of an inference engine capable of storing observed IO segments, constructing (predicting) unobserved IO segments, partitioning of an input trajectory into candidate input segments, and predicting IO trajectories. The inference engine then was interfaced with an existing LTMS to allow reasoning with the observed and hypothesized IO segments. The main modules of the LDIR were described. The software was used with the shipyard example to test the discrete-event inductive reasoner. Several sample scenarios with unique characteristics were predicted based on a few observed input/output trajectories and some user specified assumptions reflecting allowable abstractions.

To evaluate the performance of DIR and in particular LDIR, the three types of abstractions were ranked based on their compliance. Informally, I discussed how the cardinalities of hypothesized IO segments are related to one another. Then, I provided some simple fidelity assignments based on the variations of assumption set-II for different types of hypothesized IO segments. Since the DIR approach to modeling is different from the existing approaches, I continued with devising an evaluation scheme for LDIR. Two quality measures were defined. In particular, quality of a model (observed and predicted IO segments) was defined to compare alternative models. Also, quality of a simulation was defined to compare alternative predicted IO trajectories. Using these quality measures, the evaluation of the shipyard scenarios showed that LDIR predictions are as expected. I made a comparison of DIR principles with the traditional inductive modeling methodologies and pointed out that DIR meets some of the goals of a non-standard modeling methodology. In particular, LDIR was compared with SAPS, and some similarities were pointed out between LDIR and qualitative deductive modeling.

7.2 Related Research

The field of machine learning has attracted researchers from mathematics, computer science, cognitive science and engineering. Three distinct research programs have emerged during the last few decades [CM86]: (1) neural network modeling and decision theoretic techniques; (2) symbolic concept formation; and (3) knowledge intensive, domain-specific learning.

At the present time, it is impractical to engage in a comparison of the DIR methodology with existing approaches in each of the above research disciplines. This is largely due to the nature of research in inductive modeling (also called Machine Learning within the Artificial Intelligence community). Ginsberg ([Gin93], p. 300) says the following about the status of Machine Learning: "The status of learning in AI is like the status of planning, only more so: The problems are harder and recognized solutions are rarer." Thus, I have chosen to mainly discuss the research activities that are closest to what has been done here.

Nevertheless, before discussing some specific approaches which are aimed at tackling the class of systems considered here, I'll make some brief remarks about some of the main differences between the discrete-event inductive reasoning approach and statistical and neural network approaches.

Statistical approaches cannot handle abstractions in explicit form and consequently are unable to reason about any abstractions that may be made. Statistical abstractions are the means by which the certainty of a variable either is deemphasized or emphasized. In this way, some aspects of a system can be ignored altogether. In DIR, from the outset, handling of abstractions is introduced as part of the inductive modeling framework. Consequently, in statistical approaches, it is the (human) user who has to take an active role in providing the required abstractions, even those that are simple and can be handled automatically.

Neural networks approaches can have similar characteristics to statistical approaches. Moreover, their abstraction mechanisms are founded on a solid theory to a lesser extent than in the case of statistical approaches. Another distinction between DIR and these approaches is that they seem to be better equipped to handle problems with low level granularity. This advantage is mainly because much of the machinery of these approaches does not require explicit abstractions and reasoning. Presently, it is unclear how much the DIR, as described here, might suffer if it were to deal with problems that require fine grain input/output representation. The point of importance is to recognize which of the methodologies are applicable for what types of problems.

Now, I turn to AI-based approaches. The approaches addressed by the machine learning community are (deductive and inductive) generalization learning, discovery learning, and explanation-based learning. Various algorithms have been developed to handle problems that fit each of these. Of these, the inductive generalization approach appears to be the closest to the DIR approach. Prominent inductive learning algorithms are AQ11 [LM77], ID3 [Qui83], and HCV [Hon85]. Wu [Wu93] points out that these inductive learning algorithms suffer from a lack of constructive learning, incremental learning, and learning from data bases.

DIR addresses the first two — constructive and incremental learning. The DIR framework naturally supports both. The iterative IOFO specification provides a mathematical structure for dealing with creating causal relationships (or rules) between inputs and outputs. The non-monotonic reasoning approach ensures incremental learning in a well-defined fashion. This type of reasoning is more powerful than traditional rule-based approaches where rules have to be verified for proper behavior by the knowledge engineer. The combination of the IOFO representation and the non-monotonic reasoning provides explicit causality among input/output pairs and, consequently, consistency of all the input/output pairs (or the inductive model.)

This work has been concerned with systems with states. Other research efforts in inductive modeling that also have focused on such systems are the works of Biermann and Feldman [BF72], Biermann [Bie72], Cellier [Cel91], Dietterich [Die84, AD94], Gerardy [Ger89].

I already discussed in some detail how DIR and SAPS relate to each other. Now, I say a few words about the approach of Dietterich. He discusses his approach in relation to those of Biermann and Feldman [BF72], Biermann [Bie72].

Dietterich's [Die84] approach is concerned with learning about systems that have states (i.e., dynamical systems.) His approach to learning is called *Iterative Extension Method* by which partial theories are constructed successively. In this approach, the learner applies a partial theory to interpret the observed data, from which additional constraints are obtained to create another (partial) theory. A revised structure as well as new values for the state variables create an extended theory. These constraints are then used for hypothesizing additional internal state variables. The process iterates until a theory for the entire system is obtained. In [Die84], a program called EG is said to be under development. This program is to apply the iterative extension method strategy to learning about the UNIX operating system.

In concept, DIR and EG share some common goals. The EG program is said to contain two major subprograms: a reasoner and a theory formation engine. Thus, both EG and DIR use the same overall architecture. In terms of the reasoning mechanism, EG uses forward reasoning and dependency-directed backtracking of Stallman and Sussman [SS77], while DIR uses a logic-based TMS. In DIR, assumptions play a major role in predicting unobserved IO segments. EG's theory-formation engine is a means-ends analysis planner. DIR, on the other hand, uses the causality principle along with assumptions to predict IO segments. Incidentally, the name "theory-formation" is a better term than the vague "inference engine," which I have been using throughout this text. I only found this paper after I had already completed most of the writing!

Also, Dietterich studies under what conditions his method is applicable. In contrast, DIR was developed within a modeling framework. Consequently, the type of conditions that were considered by Dietterich are already contained intrinsically in the modeling formalism from the onset. Therefore, while the iterative extension method and discrete-event inductive modeling share similar ideas, they have fundamental differences. With few data, DIR is able to hypothesize output trajectories. DIR is developed from a modeling framework point of view and strives to provide a unifying framework for modeling and reasoning. The process of abstraction is a part of the inductive modeling methodology. The reasoning basis of DIR differs in significant ways from that offered by the iterative extension method.

7.3 Future Work

To continue with the work reported, two paths may be followed: applied and theoretical. Obviously, some of the following suggestions lend themselves to both applied and theoretical research. (1) Characterizing abstraction handling formally within a non-monotonic reasoning approach to provide valuable insight into the reasoning processes of DIR; (2) Studying DIR's behavior with respect to theory of Probably Approximately Correct; (3) Allowing the use of domain-specific knowledge in the reasoning process; (4) Applying other types of truth maintenance systems such as ATMS; Identifying and incorporating better means to handle multiple predictions; (5) Extending DIR to handle multi-input/output systems; (6) Including the assumption set-I in DIR, i.e., handling assumption set-I and assumption set-II simultaneously; (7) And probably most importantly, applying LDIR to other types of systems.

REFERENCES

[AD94] H. Almuallim and T.G. Dietterich. "Learning boolean concepts in the presence of many irrelevant features". *Artificial Intelligence*, 69(1–2):279–305, 1994.

[All84] J. Allen. "Towards a general theory of action and time". *Artificial Intelligence*, 23(1):123–154, 1984.

[AZ93] T. Asahi and B.P. Zeigler. "Behavioral characterization of discrete-event systems". In *AI, Simulation and Planning in High-Autonomy Systems*, pages 127–132, Tucson, AZ, Sept 1993. IEEE/CS Press.

[Ben83] J.V. Benthem. *The Logic of Time.* Kulwer Academic Publishers, Dordrecht, 1983.

[Bes89] P. Besnard. *An Introduction to Default Logic.* Springer-Verlag, New York, 1989.

[BF72] A. W. Biermann and J. A. Feldman. "On the synthesis of finite-state machines from samples of their behavior". *IEEE Transactions on Computers*, C-21:592–597, June 1972.

[Bie72] A. W. Biermann. "On the inference of turing machines from sample computations". *Artificial Intelligence*, 3(3):181–198, 1972.

[Bob80] D. Bobrow. "Special Volume on Non-Monotonic Reasoning". *Artificial Intelligence*, 13(1–2), 1980.

[Bob84] D. Bobrow. "Special Volume on Qualitative about Physical Systems". *Artificial Intelligence*, 24(1–3), 1984.

[Bre91] G. Brewka. *Nonmonotonic Reasoning: Logical Foundations of Commonsense.* Cambridge University Press, Cambridge, 1991.

[Bri77] J. Bridge. *Beginning Model Theory: The Completeness Theorem and Some Consequences.* Oxford University Press, Oxford, 1977.

[Bur90] J. Burbridge. *Within Reason: A Guide to Non-Deductive Reasoning.* Broadview Press, Lewiston, NY, 1990.

[BZ87] P. Langley H.A. Simon G.L. Bradshaw and J.M. Zytkow. *Scientific Discovery: Computational Explorations of the Creative Processes.* MIT Press, Cambridge, 1987.

[BZR89] F.E. Cellier B.P. Zeigler and J.W. Rozenblit. "Design of a simulation environment for laboratory management by robot organizations". *Journal of Intelligent and Robotic Systems*, 1:299–309, 1989.

[Cel91] F. E. Cellier. *Continuous System Modeling.* Springer-Verlag, New York, 1991.

[CK90] C.C. Chang and H.J. Keisler. *Model Theory.* North-Holland, Amsterdam, 1990.

[CL73] C.L. Chang and R.C. Lee. *Mechanical Theorem Proving.* Academic Press, New York, 1973.

[Cla78] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[CM83] R.S. Michalski J.G. Carbonell and T.M. Mitchell, editors. *Machine Learning: An Artificial Approach*, volume 1. Tioga Publishing Company, Palo Alto, 1983.

[CM86] R.S. Michalski J.G. Carbonell and T.M. Mitchell, editors. *Machine Learning: An Artificial Approach*, volume 2. Morgan Kaufmann, Los Altos, 1986.

[CZ94] A.C. Chow and B.P. Zeigler. "Parallel DEVS: A parallel hierarchical, modular modeling formalism. In *Winter Simulation Conference*, Orlando, Florida, 1994. IEEE/CS Press.

[Dav84] R. Davis. "Diagnostic reasoning based on structure and behavior". *Artificial Intelligence*, 24(3):347–410, 1984.

[Dav90] E. Davis. *Representation of Commonsense Knowledge.* Morgan Kaufmann, San Mateo, CA, 1990.

[Dem68] A.P. Dempster. "A generalization of bayesian inference". *Journal of the Royal Statistical Society, Series B*, 30(2):205–247, 1968.

[Die84] T. G. Dietterich. "Learning about systems that contain state variables". In *AAAI*, pages 96–100. William Kaufmann, 1984.

[dK86a] J. de Kleer. "An assumption based truth maintenance system". *Artificial Intelligence*, 28(2):127–162, 1986.

[dK86b] J. de Kleer. "An assumption-based truth maintenance system". *Artificial Intelligence*, 28(2):127–162, 1986.

[dKW87] J. de Kleer and B.C. Williams. "Diagnosing multiple faults". *Artificial Intelligence*, 32(1):97–130, 1987.

[Doy79] J. Doyle. "A truth maintenance system". *Artificial Intelligence*, 12(1):231–272, 1979.

[EH83] E.A. Emerson and J.Y. Halpern. Sometimes and Not Never Revisited: On Branching vs Linear Time (preliminary report). In *Proceedings 10th*

ACM Symposium on Principles of Programming Languages, pages 127–140, 1983.

[Elm78] H. Elmqvist. *A Structured Model Language for Large Continuous Systems.* PhD thesis, Lund Institute of Technology, 1978. Department of Automatic Control.

[End72] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[FdK93] K.D. Forbus and J. de Kleer. *Building Problem Solvers.* MIT Press, Cambridge, 1993.

[FF91] B. Falkenhainer and K. Forbus. "Compositional modeling: Finding the right model for the job". *Artificial Intelligence*, 51(1–3):95–144, 1991.

[Fis92] P.A. Fishwick. "An integrated approach to system modeling using a synthesis of artificial intelligence, software engineering and simulation methodologies. *ACM Transactions on Modelling and Computer Simulation"*, 2(4):307–330, 1992.

[Fis93] P.A. Fishwick. Simulation model design and execution. Department of Computer Science and Information Sciences, University of Florida, November 1993.

[For84] K. Forbus. "Qualitative process theory". *Artificial Intelligence*, 24(3):85–168, 1984.

[GBS93] K.P. Jantke G. Brewka and P.H. Schmitt, editors. *2nd International Workshop on Inductive Logic*, Reinhardsbrum Castle, Germany, Dec. 1993. LNCS 659, (subseries LNAI), Springer-Verlag.

[Ger89] Robert Gerardy. "New methods for identification of finite state machines". *Int. J. General Systems*, 15(3):97–112, 1989.

[Gil62] A. Gill. *Introduction to the Theory of Finite-State Machines.* McGraw-Hill Book Company, New York, NY, 1962.

[Gin87] M.L. Ginsberg. *Readings in Nonmonotonic Reasoning.* Morgan Kaufmann, San Mateo, CA, 1987.

[Gin89] M. L. Ginsberg. "A circumscriptive theorem prover". *Artificial Intelligence*, 39(2):209–230, 1989.

[Gin93] M.L. Ginsberg. *Essentials of Artificial Intelligence.* Morgan Kaufmann, San Mateo, CA, 1993.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[GN87] M. Genesereth and N.J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, San Mateo, CA, 1987.

[Gro88] S. Grossberg. *Neural Networks and Natural Intelligence.* MIT Press, Cambridge, 1988.

[GW92] F. Giunchiglia and T. Walsh. "A Theory of Abstraction". *Artificial Intelligence*, 57(2-3):323–389, 1992.

[Ham91] W.C. Hamscher. "Modeling digital circuits for troubleshooting". *Artificial Intelligence*, 51(1–3):223–227, 1991.

[Hay73] P. Hayes. "Nine Deadly Sins". In *AISB Quarterly: Newsletter of the Society of the Study of AI & Simulation [U.K.]*, pages 15–17, July 1973.

[HEO93] F.E. Cellier H. Elmqvist and M. Otter. "Object-oriented modeling of hybrid systems". In *ESS'93, European Simulation Symposium*, pages xxxi–xli, Delft, The Netherlands, October 1993.

[HH90] S.I. Hayakawa and A.R. Hayakawa. *Language in Thought and Action, Fifth Edition.* Harcourt Brace Jovanovich, Publishers, Orlando, Florida, 1990. First Edition, 1941.

[HM85] J.R. Hobbs and R.C. Moore, editors. *Formal Theories of the Commonsense World.* Ablex Publishing Corporation, Norwood, New Jersey, 1985.

[Hon85] J. Hong. "AE1: An extension matrix approximate method for the general covering problem". *Interntional Journal of Computer and Information Sciences*, 14(6):421–437, 1985.

[Kir91] D. Kirsh. "Special Volume on Foundation of Artificial Intelligence". *Artificial Intelligence*, 47(1):1–346, 1991.

[Kli69] G.J. Klir. *An Approach to General Systems Theory.* Van Nostrand, New York, 1969.

[Kli85] G.J. Klir. *Architecture of Systems Problem Solver.* Plenum Press, New York, 1985.

[Kuh62] T.S. Kuhn. *The Structure of Scientific Revolutions.* University of Chicago Press, Chicago, MI, 1962.

[Kui86] B. Kuipers. "Qualitative simulation". *Artificial Intelligence*, 29(3):289–338, 1986.

[Lah79] R.G. Laha. *Probability Theory.* John Wiley and Sons, New York, 1979.

[Lif85] V. Lifschitz. "Closed-World databases and Circumscription". *Artificial Intelligence*, 27(2):229–235, 1985.

[Lif86] V. Lifschitz. "On the satisfiability of circumscription". *Artificial Intelligence*, 28(1):17–27, 1986.

[Lif87] V. Lifschitz. Pointwise circumscription. In M.L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 179–193, San Mateo, CA, 1987. Morgan Kaufmann.

[Lif88] V. Lifschitz. Benchmark problems for formal non-monotonic reasoning, version 2.00. In M. Reinfrank J. de Kleer M.L. Ginsberg and E. Sandewall, editors, *2nd International Workshop on Non-Monotonic Reasoning*, pages 202–219, New York, June 13–15 1988. LNCS 346, Springer-Verlag.

[Lif89] V. Lifschitz. Circumscriptive theories: A logic-based framework for knowledge representation. In R.H. Thomason, editor, *Philosophical Logic and Artificial Intelligence*, pages 109–161. Kulwer Academic Publishers, Dordrecht, Netherlands, 1989.

[LM77] J. Larson and R.S. Michalski. "Inductive inference of vl decision rules". *SIGART Newsletter*, 63:38–44, June 1977.

[Lon78] P.E. London. Dependency networks as a representation for modeling in general problem solvers. Technical Report 698, Dept. of Computer Science, University of Maryland, August 6 1978.

[LP81] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Mac74] J.L. Mackey. *The Cement of the Universe: A Study of Causation*. Oxford University Press, 1974.

[McA80] D.A. McAllester. An outlook on truth maintenance. Technical Report AIM-551, MIT, 1980.

[McA90] D. A. McAllester. "Truth maintenance". In *AAAI-90*, pages 1109–1116, 1990.

[McC58] J. McCarthy. "Programs with common sense". In *Proceedings of the Symposium of the National Physics Laboratory*, volume 1, pages 77–84, London, UK, 1958. Also in Semantic Information Processing, Editor: M. Minsky; Reading in Knowledge Representation, Editors: R. Brachman and H. Levesque.

[McC62] J. McCarthy. "Computer programs for checking mathematical proofs". In *AMS Proceedings of Symposia in Pure Mathematics*, volume 5, pages 219–227, 1962.

[McC79] J. McCarthy. Ascribing mental qualities to machines. In M. Ringle, editor, *Philosophical Perspectives in Artificial Intelligence*, pages 93–118. Humanities Press, Atlantic Highlands, NJ, 1979.

[McC80] J. McCarthy. "A form of nonmonotonic reasoning". *Artificial Intelligence*, 13(1–2):27–39, 1980.

[McC84] J. McCarthy. "some expert systems need common sense". *Annals of the New York Academy of Sciences*, 426:129–137, 1984.

[McC86] J. McCarthy. "Applications of circumscription to formalizing commonsense reasoning". *Artificial Intelligence*, 28(1):89–116, 1986.

[McC90] J. McCarthy. *Formalizing Common Sense.* Ablex Publishing Corporation, Norwood, New Jersey, 1990. Collected Papers of John McCarthy on Commonsense Reasoning, edited by V. Lifschitz.

[McD76] D. McDermott. "Artificial intelligence meets natural stupidity". *SIGART Newsletter*, 57:5–9, April 1976.

[McD82] D. McDermott. "Non-monotonic logic II: Non-monotonic modal theories". *Journal of the Association for Computing Machinery*, 29(1):33–57, 1982.

[McD91] D. McDermott. "A general framework for reason maintenance". *Artificial Intelligence*, 50(1):289–329, 1991.

[MD80] D. McDermott and J. Doyle. "Non-monotonic logic I". *Artificial Intelligence*, 13(1–2):41–72, 1980.

[Mey85] A. Meystel. "Intelligent Control: A Sketch of the Theory". *J. Intelligent and Robotic Systems*, 2(2–3):97–107, 1985.

[MEZ89] T.I. Oren M.S. Elzas and B.P. Zeigler, editors. *Modelling and Simulation Methodology: Knowledge Systems' Paradigms.* Elsevier Science Publishers B.V. (North-Holland), Amsterdam, The Netherlands, 1989.

[MH69] J. McCarthy and P. Hayes. "Some philosophical problems from the standpoint of artificial intelligence". In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969.

[Min65] M. Minsky. "Models, minds, machines". In *Proceedings of IFIP Congress*, pages 45–49, 1965.

[MK83] R.S. Michalski and Y. Kodratoff, editors. *Machine Learning: An Artificial Approach*, volume 3. Morgan Kaufmann, Los Altos, 1983.

[Moo85] R.C. Moore. "Semantical considerations on nonmonotonic logic". *Artificial Intelligence*, 25(1):75–94, 1985.

[MR91] J.P. Martin and M. Reinfrank, editors. *Truth Maintenance Systems*, Stockholm, Sweden, August 6 1991. ECAI-90 Workshop, Springer-Verlag.

[MRS88] M.L. Ginsberg M. Reinfrank, J. de Kleer and E. Sandewall, editors. *2nd International Workshop on Non-Monotonic Reasoning*, New York, June 13–15 1988. LNCS 346, Springer-Verlag.

[MT75] M.D. Mesarovic and Y. Takahara. *General Systems Theory: Mathematical Foundations.* Academic Press, New York, 1975.

[MT89] M.D. Mesarovic and Y. Takahara. *Abstract System Theory.* Springer-Verlag, New York, 1989.

[MT93] V.W. Marek and M. Truszcynski. *Nonmonotonic Logic: Context-Dependent Reasoning.* Springer-Verlag, New York, 1993.

[MW91] Z. Manna and S.R. Waldinger. Monotonicity properties in automated deduction. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation:Papers in Honor of John McCarthy*, pages 261–280. Academic Press, New York, 1991.

[Nil80] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, 1980.

[Nil86] N.J. Nilsson. "Probabilistic logic". *Artificial Intelligence*, 28(1):71–87, 1986.

[Nil91] N.J. Nilsson. "Logic and artificial intelligence". *Artificial Intelligence*, 47(1–3):31–56, 1991.

[NR90] W. Marek A. Nerode and J. Remmel. "Nonmonotonic rule systems I". *Annals of Mathematics and Artificial Intelligence*, 1:241–273, 1990.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, CA, 1988.

[Pol75] J.L. Pollock. *Knowledge and Justification.* Princeton University Press, Princeton, New Jersey, 1975.

[Pri80] I. Prigogine. *From Being to Becoming: Time and Complexity in the Physical Sciences.* W. H. Freeman, San Francisco, 1980.

[Qui83] J.R. Quinlan. Learning efficient classification procedures and their applications to chess and games. In R.S. Michalski and J. Carbonell, editors, *Machine Learning: An Artificial Approach*, pages 463–482. Morgan Kaufmann, Los Altos, 1983.

[RdK87] R. Reiter and J. de Kleer. "Foundations of assumption-based truth maintenance systems: Preliminary report". In *Sixth National Conference on Artificial Intelligence*, pages 183–188, 1987.

[Rei78] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

[Rei80] R. Reiter. "A logic for default reasoning". *Artificial Intelligence*, 13(1-2):81–132, 1980.

[Rei87] R. Reiter. "A theory of diagnosis from first principles". *Artificial Intelligence*, 32(1):57–95, 1987.

[RH84] G.D. Ritchie and F.K. Hanna. "AM: A case study in AI methodology". *Artificial Intelligence*, 23(1):249–268, 1984.

[Ric83] E. Rich. *Artificial Intelligence.* McGraw-Hill, New York, NY, 1983.

[Ric89] T. Richards. *Clausal Form Logic: An Introduction to the Logic of Computer Reasoning.* Addison Wesley, New York, 1989.

[RKA69] P.L. Falk R.E. Kalman and M.A. Arbib. *Topics in Mathematical System Theory.* McGraw-Hill, New York, 1969.

[Rob65] J.A. Robinson. "A machine-oriented logic based on the resolution principle". *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

[Rob79] J.A. Robinson. *Logic: Form and Function.* North-Holland, New York, NY, 1979.

[RR93] R.L. Grossman A. Nerode A.P. Ravn and H. Rischel, editors. *Hybrid Systems.* LNCS 736. Springer-Verlag, 1993.

[Sha79] G.A. Shafer. *Mathematical Theory of Evidence.* Princeton University Press, Princeton, New Jersey, 1979.

[She84] S.C. Sheperdson. "Negation as failure: A comparison of clark's completed data bases and reiter's closed world assumption". *Logic Programming*, 1(1):51–79, 1984.

[Sho76] E.H. Shortliffe. *MYCIN: Computer-based Medical Consultation.* American Elsevier, NY, New York, 1976.

[Sho88] Y. Shoham. *Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence.* MIT Press, Cambridge, 1988.

[SS77] R. Stallman and G.J. Sussman. "Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis". *Artificial Intelligence*, 9(1):135–196, 1977.

[Ste90] G.L. Steel. *Common Lisp: The Language, Second Edition.* Academic Press, Digital Press, 1990.

[Sto73] H.S. Stone. *Discrete Mathematical Structures and Their Applications.* Science Research Associates, Inc., Chicago, 1973.

[Sup73] P. Suppes. *A Probabilistic Theory of Causation.* North-Holland, 1973.

[Val84] L.G. Valiant. "A theory of the learnable". *Communications of ACM*, 27:1134–1142, 1984.

[Wel86] D. Weld. "The use of aggregation in causal simulation". *Artificial Intelligence*, 30(1):1–34, 1986.

[Wel92] D. Weld. "Reasoning about model accuracy". *Artificial Intelligence*, 56(2–3):255–300, 1992.

[Wu93] Xindong Wu. "Inductive learning: Algorithms and frontiers". *Artificial Intelligence Review*, 7(3):93–108, 1993.

[Wym67] W.A. Wymore. *A Mathematical Theory of Systems Engineering : the Elements.* Wiley Series on Systems Engineering and Analysis, New York, 1967.

[Wym93] W.A. Wymore. *Model-based Systems Engineering: An Introduction to the Mathematical Theory of Discrete Systems and to the Tricotyledon Theory of System Design.* CRC, Boca Raton, 1993.

[Zad75] L.A. Zadeh. "Fuzzy logic and approximate reasoning". *Syntheses*, 3:407–428, 1975.

[Zad81] L.A. Zadeh. "PRUF A meaning representational language for natural languages". In E. Mamdani and B. Gains, editors, *Fuzzy Reasoning and Its Applications.* Academic Press, NY, New York, 1981.

[ZD79] L.A. Zadeh and C.A. Desoer. *Linear System Theory.* Krieger Publishing Company, Huntigton, New York, 1979. Original Edition, 1962, McGraw-Hill Book Company.

[Zei76] B.P. Zeigler. *Theory of Modeling and Simulation.* John Wiley and Sons, New York, 1976.

[Zei84] B.P. Zeigler. *Multi-Facetted Modelling and Simulation.* Academic Press, New York, 1984.

[Zei90] B.P. Zeigler. *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems.* Academic Press, New York, 1990.