MODELLING AND SIMULATION FOR HIGH AUTONOMY SYSTEMS

by

Sungdo Chi

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY
WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 1

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

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

entitled MODELLING AND SIMULATION FOR HIGH AUTONOMY SYSTEMS

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

[illegible]	9/5/91
F.E. Cellier	Date
[illegible]	9/5/91
J.W. Rozenblit	Date
[illegible]	9/5/91
L.C. Schooley	Date
	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.

[illegible]	10/1/91
Dissertation Director F.E. Cellier	Date

STATEMENT BY AUTHOR

SIGNED: Chi Sung Do

ACKNOWLEDGEMENTS

I would like to express my gratitude to Professor François E. Cellier and Professor Bernard P. Zeigler, my co-advisors, for their invaluable guidance, assistance, and support during my research period. I would also like to thank Professor Jerzy W. Rozenblit and Professor Larry C. Schooley, who served on my major committee, for their suggestions and help in completing my degree requirements. I appreciate all their time and effort.

I wish to express my feeling of thankfulness to Drs. Chang-Yeob Park and Sang-Hee Park at Yonsei University, Korea, who encouraged me to come to The United States to study Computer Engineering while I completed my undergraduate and M.S. degree under their direction.

There are many other people who deserve special thanks. Dr. Tag-Gon Kim has been a good friend and colleague who was always objective when I turned to him for advice. Mr. Tae-Ho Cho, Mr. Cheng-Jye Luh, Miss. Qingsu Wang, Mr. Jin-Woo Kim, Mr. Ji-Seung Nam, and many others also have been very helpful. I appreciate all their friendship and assistance.

My sincere appreciation goes to my dear wife, Soon-Young. Without her assistance, patience, and understanding, I would not have been able to pursue my career. My son, Min-Su, has supported my efforts with forbearance and goodwill. Finally, I would like to express my appreciation to my parents, especially to my mother who has passed away recently, and my wife's parents for their emotional support. To them, my special thanks.

This research was supported by NASA-Ames cooperative agreement No. NCC 2-525, “A Simulation Environment for Laboratory Management by Robot Organizations.” The support is gratefully acknowledged.

TABLE OF CONTENTS

page

LIST OF ILLUSTRATIONS

ABSTRACT

The basic objective of this research is to develop an architecture for systems capable of highly autonomous behavior by combining decision (intelligence), perception (sensory processing), and action (effector) components. The major challenge of this dissertation is the integration of high-level symbolic models with low-level dynamic (control-theoretic) models into a coherent model base.

The systematic inclusion of dynamic and symbolic models each dedicated to support a single function such as planning, operations, diagnosis or perception allows us to extend existing multi-layered control and information architectures. A knowledge-based simulation environment is employed to simulate and verify the proposed integrated model-based architecture.

The constructed working simulation version of an autonomous robot-managed laboratory demonstrates the use of multiple model families for experiment planning and execution. Tools to support the development and integration of such model families are also developed. The developed model-based architecture is elaborated by incorporating time-based simulation and causal propagation model families supporting diagnosis, repair, and replanning. This involves tools to automatically extract such models from more detailed dynamic models and structural knowledge.

Systems with high levels of autonomy are critical for unmanned, and partially manned, space missions. The utility of the proposed high autonomy system will be demonstrated with models of a robot-managed fluid handling laboratory for International Space Station Freedom to be used for research in life sciences, microgravity sciences, and space medicine. NASA engineers will be able to base designs of intelligent controllers for such

systems on the architecture developed in this dissertation. They will be able to employ our tools and simulation environment to verify such designs prior to their implementation.

CHAPTER 1
INTRODUCTION

1.1. Motivation and Goals

Emerging from the control field, *intelligent control* is viewed as a new paradigm for solving control problems [58],[59]. Its relatively narrow interpretation of the "control problem" does not include, for example, the control needed by a system to diagnose and repair itself after significant insults to its physical structure. However, requiring greater degrees of autonomy from a system forces a more expanded view. Still within the control perspective, Antsaklis *et al.* [6] developed a more inclusive framework for autonomous systems. This dissertation contains an extensive bibliography on intelligent and autonomous control. To achieve autonomy, artificial intelligence is one, among other, means to this end.

Although, criteria for artificial intelligence [27],[64] implicitly include autonomy, robotics research is realizing that autonomy requires the integration of AI decision making components together with perception and action components [45]. Even some AI researchers state that problems in AI itself might be better solved by incorporating solutions from perception and motor control [9].

To include such diverse systems as planetary colonies, self-operating factories, and telerobotic laboratories along with autonomous land and space vehicles in the autonomy umbrella, we employ the following definition [63]:

Autonomy is the ability to function as an independent unit or element over an extended period of time, performing a variety of actions necessary to achieve pre-designated objectives while responding to stimuli produced by integrally contained sensors.

Saridis [78] developed a three layer hierarchy (execution, coordination and management) for intelligent control [78] which is supposed to reflect increasing intelligence with decreasing precision. Antsaklis *et al.* [6] refine the hierarchy to an arbitrary number of layers, depending on the particular application. The coupling of control and information at various layers characterizes the framework recently proposed by Albus [1],[2],[3]. Generalizing the earlier NASREM framework, Albus elaborates an architecture of seven levels, each of which has its own Task Decomposition and Execution, Sensory Processing, World Model, Global Memory, and Value Judgement components. Components at one level communicate with each other and with corresponding components at higher and lower levels. The Albus architecture is the most general and elaborate attempt to integrate perception, decision, and action to achieve intelligence/autonomy. However, it leaves many details unspecified, particularly of interest here, the role of models.

The alternative architecture for autonomous robotics developed by Brooks [8] does not explicitly incorporate higher level planning and world model perception. This approach by-passes the computational burden required in the multi-layered architectures. Although achievements such as robotic "insects" are impressive, it is not clear whether such an approach "scales up" to systems that for example, operate sophisticated equipment and are able to detect, diagnose, and repair faults in such equipment. However, the Brooks alternative makes clear that one major hurdle to overcome in the architectural concepts of Albus, Saridis, Antsaklis, etc. is the heavy on-line computation times needed for higher level functions such as planning and diagnosing. The proposed research seeks to reduce such on-line computation by off-line pre-compilation to produce simplified models individually matched to the tasks needing to be performed. Such models are to be attached to generic engines that can interpret them efficiently with reduced processing times.

Models are directly associated with the tasks that are needed to achieve system objectives. Some of the tasks supported by models are listed under the three primary control layers [78]:

- *management*
  - *action planning (plan development, plan evaluation)*
  - *FDIR (fault detection, diagnosis, isolation, and repair)*
- *coordination*
  - *path planning*
  - *scheduling*
  - *control*
  - *FDIR*
- *execution*
  - *object identification*
  - *manipulation*
  - *navigation*
  - *operation*
  - *FDIR*

In the proposed *model-based architecture*, knowledge is encapsulated in the form of models that are employed at the various control layers to support the predefined system objectives. The term "model" here refines the "world model" concept: "the system's best estimate of objective reality" [1]. It recognizes that an autonomous system must maintain models in a variety of formalisms and at various levels of abstraction. Lower control layers are more likely to employ conventional differential equation models with symbolic models more prevalent at higher layers. A key requirement is the systematic development and integration of dynamic and symbolic models at the different layers. In this way, traditional control theory, where it is applicable, can be interfaced with AI techniques, where they are necessary. Antsaklis *et al.* [6] propose that so-called hybrid modelling methodology must be adopted --- the combination of traditional differential/difference equation models with more recent symbolic formalisms [108]. The proposed research is to develop concepts and tools to facilitate such hybrid methodology.

1.2. Basic Method

Note that an autonomous system could, in principle, base various functional aspects such as operation, diagnosis, repair, planning, and other activities on a single comprehensive model of its environment. However, such a model would be extremely unwieldy to develop and lead to intractable computations in practice. Instead, our architecture employs a multiplicity of partial models to support system objectives. As indicated, such models differ in level of abstraction and in formalism. The partial models, being oriented to specific objectives, should be easier to develop and computationally tractable. However, this approach leads to sets of overlapping and redundant representations. Concepts and tools are needed to organize such representations into a coherent whole. Structure and behavior preserving morphisms from model theory [108],[109],[115] can connect models at different levels of abstraction so that they can be developed to be consistent with each other and can be consistently modified.

There are three major formalisms, or short-hands, for dynamic system specification. Differential and difference equations employed to describe continuous systems have a long history of development whose mathematical formalization came well before the advent of the computer. Discrete event modelling is of more recent vintage and is finding ever more application to analysis and design of complex manufacturing, robotic, communication, and computer systems, among others [31],[41]. In contrast to continuous systems simulation, discrete event simulations were made possible by, and evolved with, the growing computational power of computers. Formalization of the models that underlay discrete event simulations is therefore a relatively new concern [61],[115]. Formalisms that can capture both discrete and continuous behavior are also of increasing importance [62],[69],[89].

The classical AI knowledge representation schemes, such as rules, semantic nets, and frames, can be viewed as essentially symbolic formalisms for static models. They provide organizations for large bodies of facts with mechanisms for inferencing, access path development, and comparison based on symbol manipulation. New directions in AI, for example qualitative modelling and planning, are finding it necessary to take time into account [5],[88].

On the other hand, while differential/difference equation formalisms are numeric in character, this is not necessarily true of more recent dynamic systems formalisms. Discrete event and automata models, for example, usually include state representations of a symbolic nature. Antsaklis *et al.* [6] refer to discrete event models that do not include a time base explicitly [116] as "logical," thereby emphasizing their symbolic nature. Narain [62] has developed a logic-based formalism intended to facilitate reasoning about the behavior of combined discrete/continuous models.

Qualitative physics has done much to bring qualitative formalisms for dynamic model specification to attention. AI researchers in qualitative physics envision an ideal formalism which captures commonsense knowledge of dynamic systems without the use of differential equations [24],[44],[52],[53],[70]. The ability of such models to deal with incomplete specification is emphasized, although the cost is heavy ambiguity and trajectory branching. Model-based diagnosis [25],[33],[44],[71] is a related approach that, using models, seeks to support deeper reasoning than possible in conventional "shallow" expert system approaches. The diagnostic models in the proposed research will be of both shallow and deep nature [80] derived from underlying discrete event specifications.

Traditional discrete event formalisms incorporate uncertainty in the form of stochastic processes. Modern views recognize types of uncertainty that cannot be represented in a probabilistic manner such as ambiguity, lack of relevant factual knowledge, and deliberate use of linguistic imprecision. Fuzzy set theory is the predominant framework for representing such uncertainty (although it is not the only one). Incorporating fuzzy set mechanisms into models is drawing increasing attention [18],[28].

For autonomous systems design, any, and all, of such formalisms are potentially applicable. Conventional automatic control theory makes heavy use of differential/difference equation models to capture continuous interaction with the world at the lower physical level, where "physical" includes aerodynamic, chemical, biological, ecological, and other processes. Planning and scheduling of manufacturing and other operations rely on simulation of discrete event models. It is reasonable to seek to achieve high autonomy by retaining such existing model formalisms and associated simulation engines as a basis, introducing newer symbolic and qualitative formalisms as needed.

The event-based control methodology [102],[109] provides an approach to process and device control that is especially tuned to the constraints of high-autonomy applications. It provides a relatively simple and robust real-time control layer that can be linked to higher level symbolic reasoning layers with the abstraction approach.

1.3. Organization of the Dissertation

Chapter 2 briefly reviews the System Entity Structure / Model Base framework as a basis of the model-based approach. The specification of the system entity structure and the concept of model synthesis are presented. In Chapter 3, the discrete event system specification formalism (DEVS) and its implementation in an object-oriented environment are introduced as a basic tool to analyze and design complex systems. Chapters 2 and 3 review basic concepts that have been previously described by other authors. They are included in this dissertation solely for the purpose of enhancing the understandability of subsequent chapters without forcing the reader to digest other documents first. Chapter 4 shows an extension to the DEVS formalism that facilitates the symbolic representation of event times by extending the time base from real numbers to the field of linear polynomials over real numbers. This chapter also proposes a linear polynomial constraint checking algorithm for efficient management of non-determinism in symbolic simulation. In Chapter 5, a robot-managed laboratory of the future is introduced as an example of a high autonomy system design. This chapter shows the construction of a space-borne laboratory environment by means of an entity structure knowledge representation. Chapter 6 presents the concepts of model-based architectures for high autonomy systems as an extended paradigm that subsumes both control and AI paradigms. Chapter 7 presents the concept of DEVS-based intelligent control. This chapter then shows the intelligent control of a space-adapted fluid mixing example. The relation with other programmable logic controller (PLC) approaches is also discussed. In Chapter 8, the development of a model-based task planning system is presented by showing a coherent way of integrating typical planning paradigms such as "searching" and "representation." This chapter proposes a task planning methodology using the system entity structure concept. Chapter 9 presents a hierarchical model-based diagnosis for inferring single or multiple faults from the knowledge of faulty behaviors of component models and their causal structures. The chapter shows a local and a global reasoning concepts to extract cause-effect tables from the faulty component models using symbolic simulation. Chapter 10 presents a systematic integration of the previously developed methodologies and architectures into a coherent whole. This chapter then proposes a synthesis methodology for automatically generating an autonomous system architecture. Chapter 11 concludes the dissertation by summarizing the results, showing the main contributions, and suggesting further research related to this work.

CHAPTER 2
SYSTEM ENTITY STRUCTURE / MODEL BASE

2.1. Overview

The *System Entity Structure / Model Base (SES/MB)* framework was proposed by Zeigler [107],[109],[112] as a step toward interfacing the dynamic-based formalism of simulation with the symbolic formalism of AI. It consists of two components: a *system entity structure* and a *model base*. The *system entity structure*, declarative in character [48],[50], represents the knowledge of decomposition, component taxonomies, coupling specification, and constraints. The *model base* contains models that are procedural in character, expressed in dynamic and symbolic formalisms.

2.2. System Entity Structure.

The System Entity Structure (SES) is a representation scheme that contains the decomposition, coupling, and taxonomy information of a system [107],[112]. Formally, the SES is a labeled tree with attached variable types that satisfies six axioms - uniformity, strict hierarchy, alternating mode, valid brothers, attached variables, and inheritance. A detailed description of the axioms is given in [109],[112].

The SES contains three types of nodes - entities, aspects, and specializations - which represent three types of knowledge about systems. The entity node, which may have several aspects and/or specializations, corresponds to a model component that represents a real world object. The aspect node (denoted by a single vertical line in the labeled tree of Figure 2.1) represents one decomposition, out of many possible, of an entity. Thus, the children of an aspect node are entities, distinct components of the decomposition. The specialization node (denoted by a double vertical arrow in the labeled tree of Figure 2.1) represents a way in which a general entity can be specialized into entities.

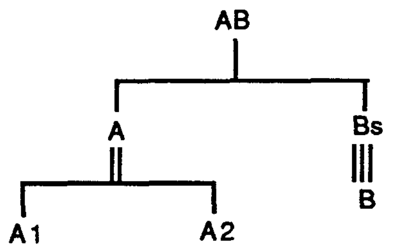


Figure 2.1. An Example of a System Entity Structure

A multiple entity is an entity that represents a collection of homogeneous components. We call such components a multiple decomposition of the multiple entity. The aspect of such a multiple entity is called multiple aspect (denoted by a triple vertical line in the labeled tree of Figure 2.1). Note that instead of presenting all Bs for B's components, only one B is placed in the labeled tree.

Pruning extracts a sub-structure of the SES by selecting one aspect and/or one specialization for each entity in the SES. The pruning operation also expands multiple entities as well as values of attributes attached to entities in the SES.

The SES has been realized in the Scheme LISP environment. The realization called ESP-Scheme was described in [48]. ESP-Scheme has been used as a means of representing structural knowledge for a simulation model in knowledge-based modelling/simulation research.

2.3. SES Pruning and Model Synthesis

One application of the SES/MB framework relates to the design of systems [75],[82]. Here the SES serves as a compact knowledge representation scheme of organizing and generating the possible configurations of a system to be designed. To generate a candidate design, we use a process called *pruning* which reduces the SES to a so-called *pruned* *entity structure* (PES). Such structures are derived from the governing structure by a process of selecting from alternatives wherever such choices are presented. Not all choices may be selected independently. Once some alternatives are chosen, some options become unavailable while others are enabled. Moreover, rules may be associated with the entity structure that further reduce the set of configurations to be considered [42]. In the model-based architecture of high autonomy systems, the pruning is an initial planning process supporting the selection of the sequenced/planned model.

As shown in Figure 2.2, pruned entity structures can be stored along with the SES in files forming *the entity structure base.*

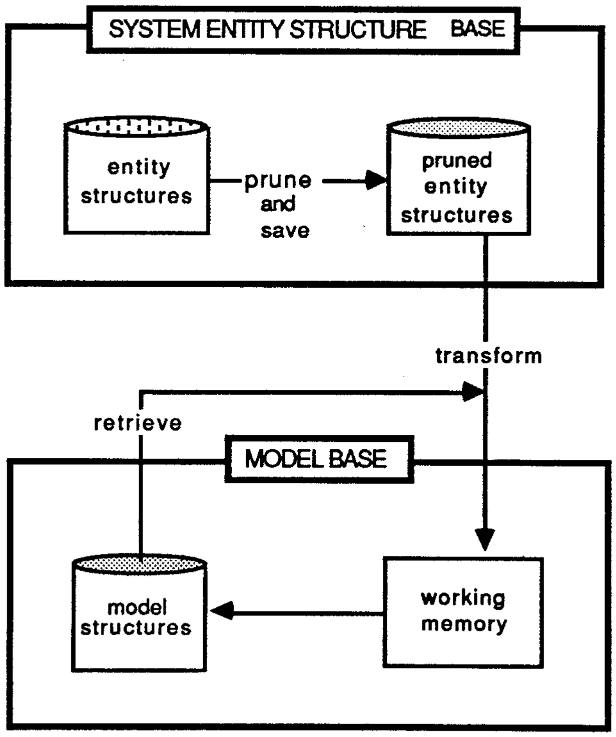


Figure 2.2. The System Entity Structure/Model Base (SES/MB) Environment

Hierarchical simulation models may be constructed by applying the *transform* function to pruned entity structures in working memory. As it traverses the pruned entity structure, *transform* calls upon a retrieval process to search for a model of the current entity. If a model is found, it is used, and the transformation of the entity subtree is aborted. *Retrieve* looks for a model first in working memory. As it traverses the pruned entity structure, *transform* calls upon a retrieval process to search for a model of the current entity. If no model is found in working memory, the *retrieve* procedure searches through model definition files, and finally, provided that the entity is a leaf of the currently visited SES, it checks other pruned entity structure files. If another PES is found that defines the currently investigated leaf entity as its root entity, a new incarnation of the *transform* process is spawned to construct a model of that leaf entity. Once this construction is complete, the main *transform* process is resumed.

The result of a transformation is a model expressed in an underlying simulation language such as DEVS-Scheme [109], ready to be simulated and evaluated relative to the modeler's objective. The fact that the *transform* process can look for previously developed pruned entity structures, in addition to basic model files, has an important consequence for reusability.

2.4. Hierarchical Reusability of Pruned Entity Structures

The power to generate a large collection of models brings with it the need to archive the collection for reuse. Think of the SES as a powerful "printing press" and the collection of PESs the "books" to be archived. Properly cataloguing the books rolling of the press will help us find existing books that meet our current needs. The situation is described in Figure 2.3. A PES uniquely represents a model in that the PES contains all the information needed by the transformation process to synthesize it. We assume that the model base is complete in the sense that it contains models for all the atomic entities referenced in the PESs.

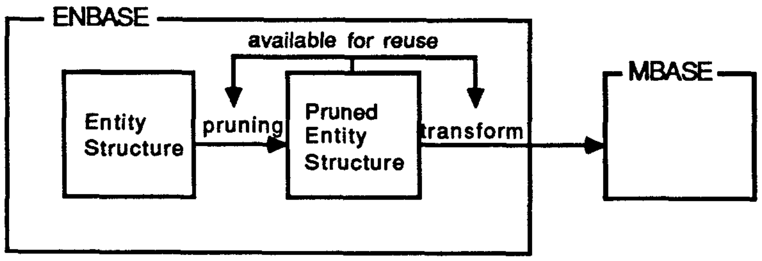


Figure 2.3. Reusability of Pruned Entity Structures

The basic requirement of cataloguing is easy to state: a PES should be catalogued in such a way as to facilitate subsequent recognition and retrieval. Extensions of PESs can be used to identify different prunings that all have the same root and therefore represent different models of the same entity.

The next requirement for reusability is that when a leaf entity is encountered in the pruning procedure, such versions automatically be available for user selection as shown in Figure 2.4. This captures the hierarchical sense of model reusability, since existing versions of leaf entities can be "pulled of the shelf" and "plugged in" as components to a more encompassing model.

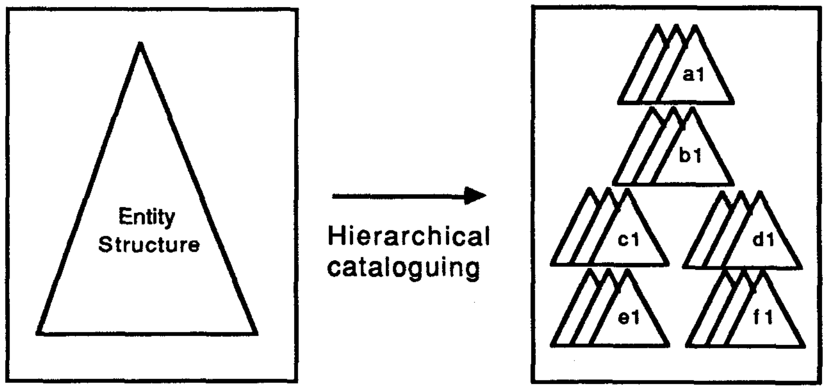


Figure 2.4. Hierarchical Pruning for Reuse

More than one SES may exist in the entity structure base, each one representing a family of models for its root entity. In principle, every entity might have its own SES but this would lead to extreme fragmentation of the encoded knowledge. Reasons for giving an entity its own SES include: the large size of its family of possible prunings, its high likelihood of being modified, and its occurrence in several places of existing SESs. The role of reusing entity structures for model-based task planning is discussed in chapter 8.

CHAPTER 3
DEVS FORMALISM AND DEVS-SCHEME

3.1. Overview

Discrete event modelling is finding ever more applications to analysis and design of complex manufacturing, communication, and computer systems, among others. Powerful languages and environments have been developed for describing such models for computer simulation [31]. Yet, the general understanding of the nature of discrete event systems *per se* is still in its infancy compared to that of continuous systems.

Differential equations employed to describe continuous systems have a long history of development whose mathematical formalization came well before the advent of the computer. In contrast, discrete event simulations were made possible by, and evolved with, the growing computational power of computers. The prime requirement for conducting such simulations was to be able to program a computer appropriately. Not of immediate utility, computer-independent model description formalisms for discrete event systems, paralleling the differential equations for continuous systems, were late in coming. Yet, it is now being recognized that our understanding of complex systems may be greatly enhanced with such mathematically based formalisms.

A system can be described as existing in any one of a set of internal configurations or "states" with discrete inputs and outputs. The state of a system summarizes the information concerning past inputs that is needed to determine the response of the system to subsequent inputs [108]. The discrete event formalism focuses on the changes of state variable values and generates time segments that are piecewise constant.

3.2. DEVS Formalism

The Discrete Event System Specification (DEVS) formalism introduced by Zeigler [108] provides a means of specifying a mathematical object called a *system.*

A DEVS (Discrete Event System Specification) is a structure:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$$

where

X is the set of external (input) event types

S is the sequential state set

Y is the output set

$\delta_{int} : S \rightarrow S$, is the internal transition function

$\delta_{ext} : Q * S \rightarrow S$, is the external transition function, where Q is the total state set

$Q = \{ (s,e) \mid s \in S, \ 0 \leq e \leq ta(s) \}$

$ta : S \rightarrow R^{+}_{0,\infty}$ is the time advance function, where the $R^{+}_{0,\infty}$ is the non-negative reals with ∞ adjoined.

$\lambda : S \rightarrow Y$, is the output function

A system has a time base, inputs, states, outputs, and functions for determining next states and outputs given current states and inputs. The formalism specifies :

1) basic models from which larger ones are built.

2) how these models are connected together in a hierarchical fashion.

A basic model, called *atomic model*, contains the following information:

• the set of input ports through which external events are received;

• the set of output ports through which external events are sent;

• the set of state variables and parameters: two state variables are usually present -- phase and sigma (in the absence of external events the system stays in the current phase for the time given by sigma);

- the internal transition function, which specifies to what next state the system will transit after the time given by a sigma has elapsed;
- the external transition function, which specifies how the system changes state when an input is received -- the effect is to place the system in a new phase and sigma, thus scheduling it for a next internal transition;
- the output function, which generates an external output just before an internal transition takes place;
- the time advance function, which controls the timing of internal transitions -- when the sigma state variable is present, this function just returns the value of sigma.

The second form of model, called *coupled model*, tells how several component models are coupled together to form a new model. This latter model can itself be employed as a component in a larger coupled model, thus giving rise to hierarchical model construction. A coupled model contains the following information:

- the set of input ports through which external events are received;
- the set of output ports through which external events are sent;
- the external input coupling, which connects the input ports of the coupled model to one or more of the input ports of the components -- this directs inputs received by the coupled model to designated component models;
- the external output coupling, which connects output ports of components to output ports of the coupled model -- thus, when an output is generated by a component, it may be sent to a designated output port of the coupled model, and be transmitted externally;
- the internal coupling, which connects output ports of components to input ports of other components -- when an output is generated by a component, it may be sent to the input ports of designated components.

3.3. DEVS-Scheme

DEVS-Scheme [109], a general-purpose modelling and simulation environment, is an implementation of the DEVS formalism in SCOOPS, an object-oriented superset of Scheme (a Lisp dialect), which enables the modeler to specify models directly in DEVS terms. DEVS-Scheme supports building models in a hierarchical, modular manner (successively synthesizing systems from component models to form new component models of a higher hierarchy level), a systems-oriented approach not supported in conventional languages.

The class specialization hierarchy of DEVS-Scheme is illustrated in Figure 3.1. All classes are subclasses of the universal class *entities*, which provides tools for manipulating objects in these classes. The inheritance mechanism ensures that such general facilities need only be defined once. *Models* and *processors*, the main subclasses of *entities*, provide the basic constructs needed for modelling and simulation, respectively. *Models* is further specified into the major classes *atomic-models* and *coupled-models,* which in turn are specialized into more specific classes. Class *processors,* on the other hand, has three specializations: *simulators*, *co-ordinators*, and *root-co-ordinators*, which serve to handle all the simulation needs.

The class *atomic-models* realizes the atomic-level of the DEVS model formalism. *Forward-models* and *table-models*, subclasses of *atomic-models*, generate models that inherit all the instance variables and methods of *atomic-models.* However, they generate more specific models based on a forward-chaining production rule paradigm and a table-lookup paradigm, respectively.

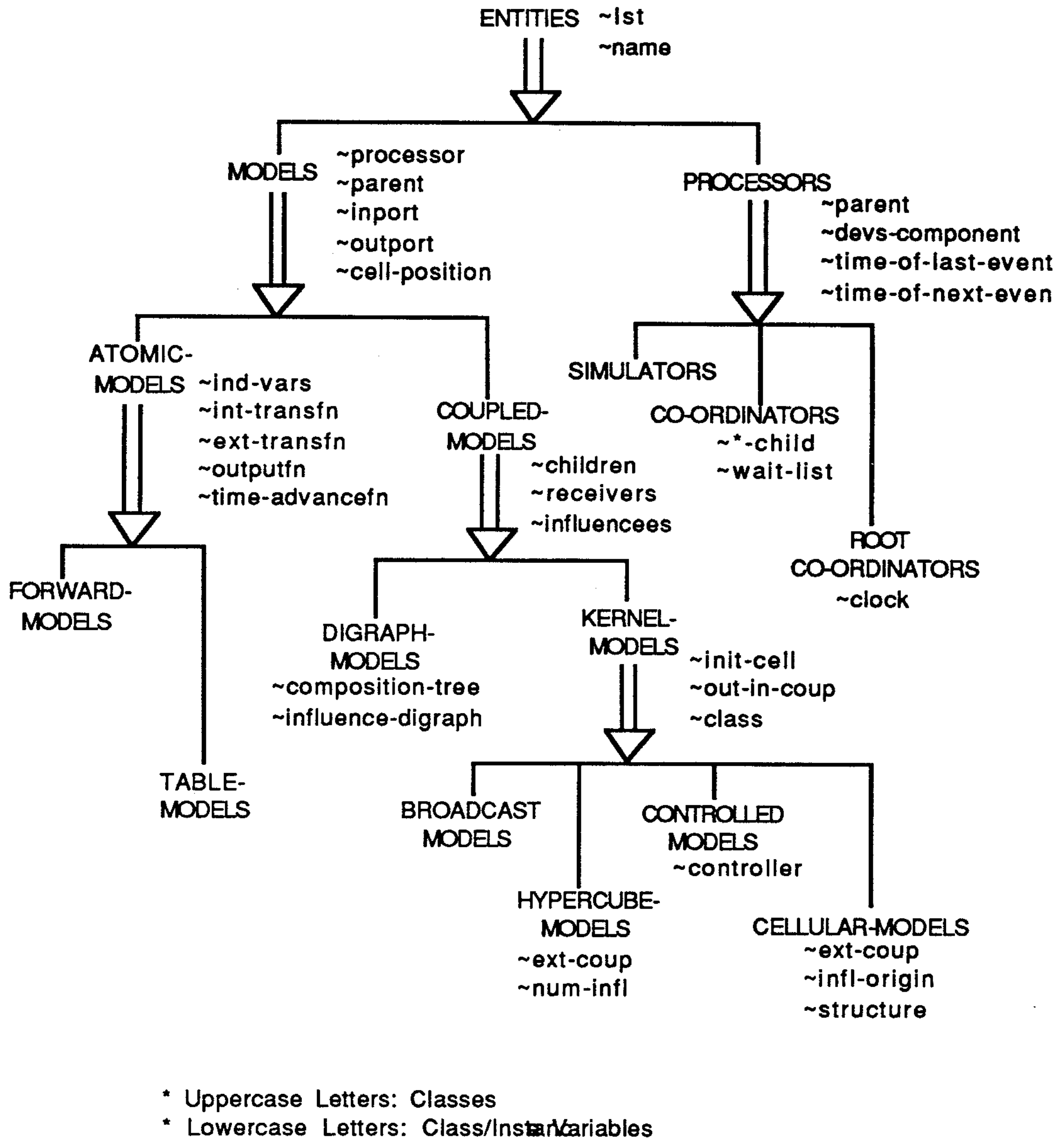

ENTITIES ~lst ~name
MODELS ~processor ~parent ~inport ~outport ~cell-position
PROCESSORS ~parent ~devs-component ~time-of-last-event ~time-of-next-even
ATOMIC-MODELS ~ind-vars ~int-transfn ~ext-transfn ~outputfn ~time-advancefn
COUPLED-MODELS ~children ~receivers ~influencees
SIMULATORS
CO-ORDINATORS ~*-child ~wait-list
ROOT CO-ORDINATORS ~clock
FORWARD-MODELS
TABLE-MODELS
DIGRAPH-MODELS ~composition-tree ~influence-digraph
KERNEL-MODELS ~init-cell ~out-in-coup ~class
BROADCAST MODELS
CONTROLLED MODELS ~controller
HYPERCUBE-MODELS ~ext-coup ~num-infl
CELLULAR-MODELS ~ext-coup ~infl-origin ~structure
* Uppercase Letters: Classes
* Lowercase Letters: Class/Instance Variables

Figure 3.1. Class Hierarchy of DEVS-Scheme

The class *coupled-models* is a major class, which embodies the hierarchical model composition constructs of the DEVS formalism. In the DEVS formalism, a coupled-model is defined by specifying its components and the coupling relations which establish the desired communication links. A detailed description of DEVS-Scheme is given in [108],[109].

The DEVS formalism is more than just a means of constructing discrete event simulation models. It provides a formal representation technique for systems that are capable of mathematical manipulation, just as differential equations serve this role for continuous systems [102].

CHAPTER 4
SYMBOLIC DEVS

4.1. Overview

Extending the discrete event modelling formalism to facilitate its symbol manipulation capabilities is important in the context of its use for intelligent control and the design of high autonomy systems. This chapter shows an extension to the DEVS formalism that facilitates the symbolic representation of event times by extending the time base from the real numbers to the field of linear polynomials over real numbers. A simulation algorithm is developed to generate the branching trajectories resulting from the underlying non-determinism. To efficiently manage symbolic constraints, we developed a consistency checking algorithm for linear polynomial constraints based on a feasibility checking algorithm borrowed from linear programming. The extended formalism offers a convenient means to conduct multiple, simultaneous explorations of model behaviors. In Chapter 9, we shall show how symbolic DEVS plays a role for fault diagnosis.

4.2. Symbolic DEVS Concept

The discrete event-based control methodology, which will be discussed in greater detail in Chapter 7, provides an approach to process and device control that is especially tuned to the needs of high autonomy applications. It provides a relatively simple and robust real-time control layer that can be linked to higher level symbolic reasoning layers with appropriate structure and behavior preserving morphisms [41]. A crucial requirement in discrete event-based control is the coupling of task execution with an error recovery capability. Such capability is based on diagnosis of faults based on a detected anomalous behavior. The development of symbolic DEVS to be discussed was motivated by the need for a modelling formalism to support such diagnostic requirements.

4.2.1. Introduction

Until now, the time base of discrete event models has been assumed to be numerical. To enhance the capability of discrete event models, we extend the DEVS formalism[108] to be defined on a symbolic time base. In conventional DEVS, and indeed, in most system formalisms [51],[57],[93], the time base and its operations and relations are performed over the field of real numbers. In symbolic DEVS, this field is extended to the linear polynomials over the reals. For example, the expression "5 + 2 * *travel_time* + *delay_time*" is a legal time value and might represent the time taken to make a round trip journey. When *travel_time* and *delay_time* are assigned numerical values, this expression evaluates to a real number. Thus precise knowledge of any event time can be readily incorporated by setting its symbol to a real value. Therefore symbolic DEVS is truly an extension of conventional DEVS and reduces to the latter when all times are real valued. This contrasts with qualitative representations of differential equation models (e.g., [52]) that move to a coarser grained formalism in which exact timing cannot be recovered with acquisition of more information.

A network or coupled model[109] of symbolic DEVS components is non-deterministic since the minimum of a set of next event times is not unique as in a conventional DEVS of the same kind. Thus trajectories split into tree-like structures. Each minimal element in a set of next event times represents a possible state transition; in making the choice of such an element, we generate a set of constraints asserting that it is less than all other minimal elements. This constraint applies to the trajectory branch just initiated. Likewise constraints are accumulated and inferenced to constrain minimal next event times in subsequent branchings. For example, asserting "$t_1 < t_2$" at one branch, and "$t_2 < t_3$" at a subordinate one, implies also that "$t_1 < t_3$" in the latter subtree. Since symbolic times are linear expressions, we adopt techniques from linear programming to efficiently manage and inference the constraint sets [15],[67].

One major application of symbolic DEVS is to perform the causal propagation required for fault diagnosis. Conventional DEVS models can handle such propagation but require fixing the event times to particular real numbers to do so. By allowing the times to be expressed symbolically, we generate a family of trajectories that represent all possible sequences of events resulting in a particular breakdown. Such a family will be employed to trace a detected anomaly back to the components that might have caused it. The symbolic extension allows for both the situations where timings are unknown, or are known, but can vary. This approach to model-based diagnosis supersedes others [25],[44],[71] in that it intrinsically represents timing effects. Dynamic models are necessary in task execution applications where effects of an anomaly occurring in some task are not apparent until they have propagated, some time later and in some transformed manner, to a detectable condition. More details will be presented in Chapter 9.

Besides being applicable to fault diagnosis, symbolic DEVS may have a variety of applications where there is a need for efficient generation of a family of trajectories characterizing all feasible scenarios of a given model. For illustration, we shall show how the rules for strikes, balls, and hits in baseball can be generated.

4.2.2. Symbolic Simulation

Often, the basic equivalent of a multicomponent DEVS is not derived by formal manipulation, but is investigated through simulation. In the case of symbolic multicomponent models, an appropriate simulation algorithm has to be developed that handles the non-determinism arising due to the symbolic nature of the underlying time set. Let $M = < S, \delta_{int}, ta >$ be a non-deterministic symbolic DEVS. The algorithm generates a branching tree structure of nodes, each representing a generated state of M. Since such an M represents the basic equivalent of a symbolic multicomponent model, the algorithm is easily extended to the latter case.

Each node is an object containing slots for:

the time of this event, t (a linear polynomial)
the chosen time advance, τ (a non-negative linear polynomial)
the accumulated constraint set, C
the prior state of the *DEVS*, s
the next state of the *DEVS*, s'

The symbolic simulation algorithm is:

Initialize the model to its desired initial state, s_0, at time, t_0. Create the root node:

time = t_0
chosen time advance = unspecified
constraint set = $\emptyset$
prior state = unspecified
next state = s_0

(Note: the constraint set may also be initialized to a non-empty set if desired. This could occur if the initial state is a terminal state of some previous run or if some *a priori* constraints need to be established for the simulation.)

Starting at the root node, skip step 1 in the following cycle, and continue with step 2 with s' taken as s_0.

From the current node with:

time = t
chosen time advance = τ
constraint set = C
prior state = s
next state = unspecified

Do the following:

```
1. Given $s$ and $\tau$, simulate to get $s' = \delta_{int}(s, \tau)$
2. Given $s'$, obtain $ta(s')$
3. If $ta(s') = \infty$, then leave this node, else:
 For each $\tau_i \in ta(s')$
   If $C' = reduce(C \cup \{\{\tau_i < \tau_j\} / j \neq i \in ta(s')\}) \neq \phi$
   then create a node with
     time = $t + \tau_i$
     time advance = $\tau_i$
     constraint set = $C'$
     prior state = $s'$
     next state = to be computed
4. Repeat the simulation from each newly created node
```

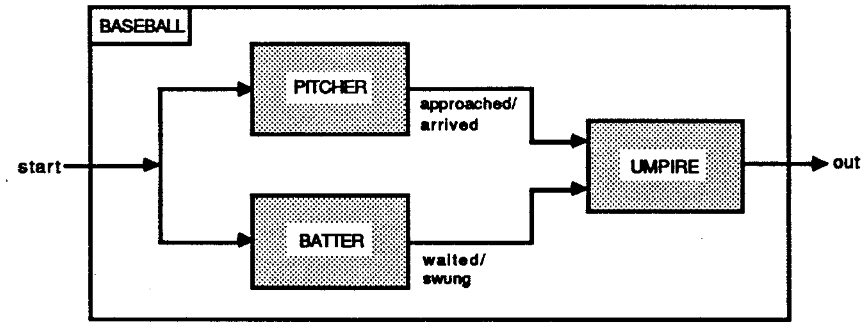
Here *reduce(C)* tests whether a set of constraints, C, is consistent; if so it returns a subset containing no redundant constraints, otherwise it returns *{ }* [15]. The *Reduce* algorithm, which was developed on the basis of a linear polynomial constraint checking algorithm, will be discussed later.

The abstract simulator implementation in DEVS-Scheme [109] has been expanded to execute such a simulation for one-level coupled-models with symbolic atomic-model components. In steps 1 and 2, it employs the existing simulation apparatus to set the initial state of the model to s. It then performs a one-step simulation to obtain the next state, s', and the next time advance set, *ta(s')*. Step 3 extracts a subset of the latter set -- the minimal set relative to the accumulated constraints C. In other words, for a particular member, $\tau_i \in ta(s')$, it checks all inequalities $\{\tau_i < \tau_j\}$, $j \neq i$ for consistency with C. This tests whether τ_i can be made to be the smallest element in *ta(s')* without violating the constraints, C. If constraint violation occurs then the assertion that τ_i can be made the smallest element is untenable. In this case, no branch is created for τ_i. Otherwise, τ_i can safely be forced to be the minimum of *ta(s')*, and a branch is created for it. Along this branch the constraint set must be updated to reflect the new set of assertions just made. *Reduce* does this while ensuring that the resulting set of constraints is irredundant.

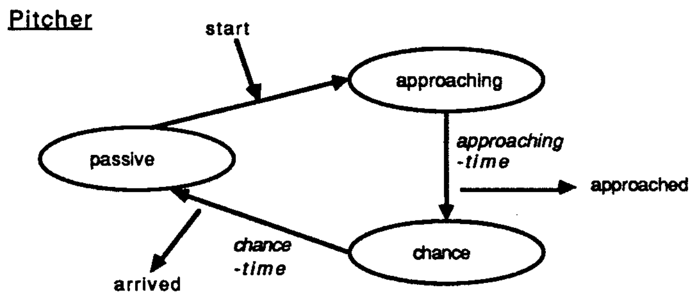
4.2.3. Example: Symbolic Simulation of Baseball

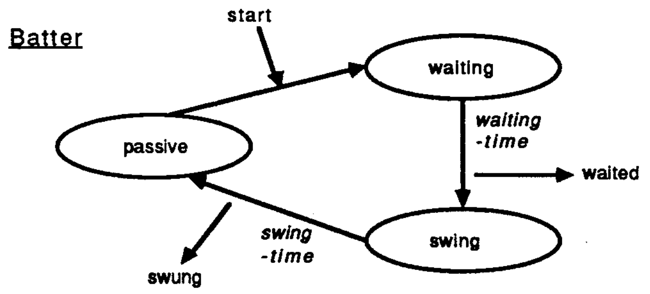
For another illustration of symbolic simulation, consider some timing relations between pitcher and batter in baseball. The ball departs from the pitcher and approaches the home plate providing an opportunity for the batter to swing. If not hit, it reaches the catcher. On the other hand, the batter waits for the ball to reach a good position and then swings to hit the ball. Figure 4.1a) shows a coupled model, and Figure 4.1b) shows the symbolic DEVS state transition graphs of its PITCHER and BATTER as components.

When the PITCHER component receives the *start* command, it changes the state from *passive* to *approaching*. After *'approaching-time* has elapsed, it outputs *approached* to the UMPIRE and changes its state to *chance* during *'chance_time*. Then it goes to the *passive* state after sending *arrived* to the UMPIRE. The BATTER has basically the same state transition, but different states and times. Symbolic simulation generates the tree of trajectories in Figure 4.1c), which are illustrated in Figure 4.1d). Note that the leaves of Figure 4.1c) represent all feasible outcomes of the model together with the timing relationships necessary to generate them.

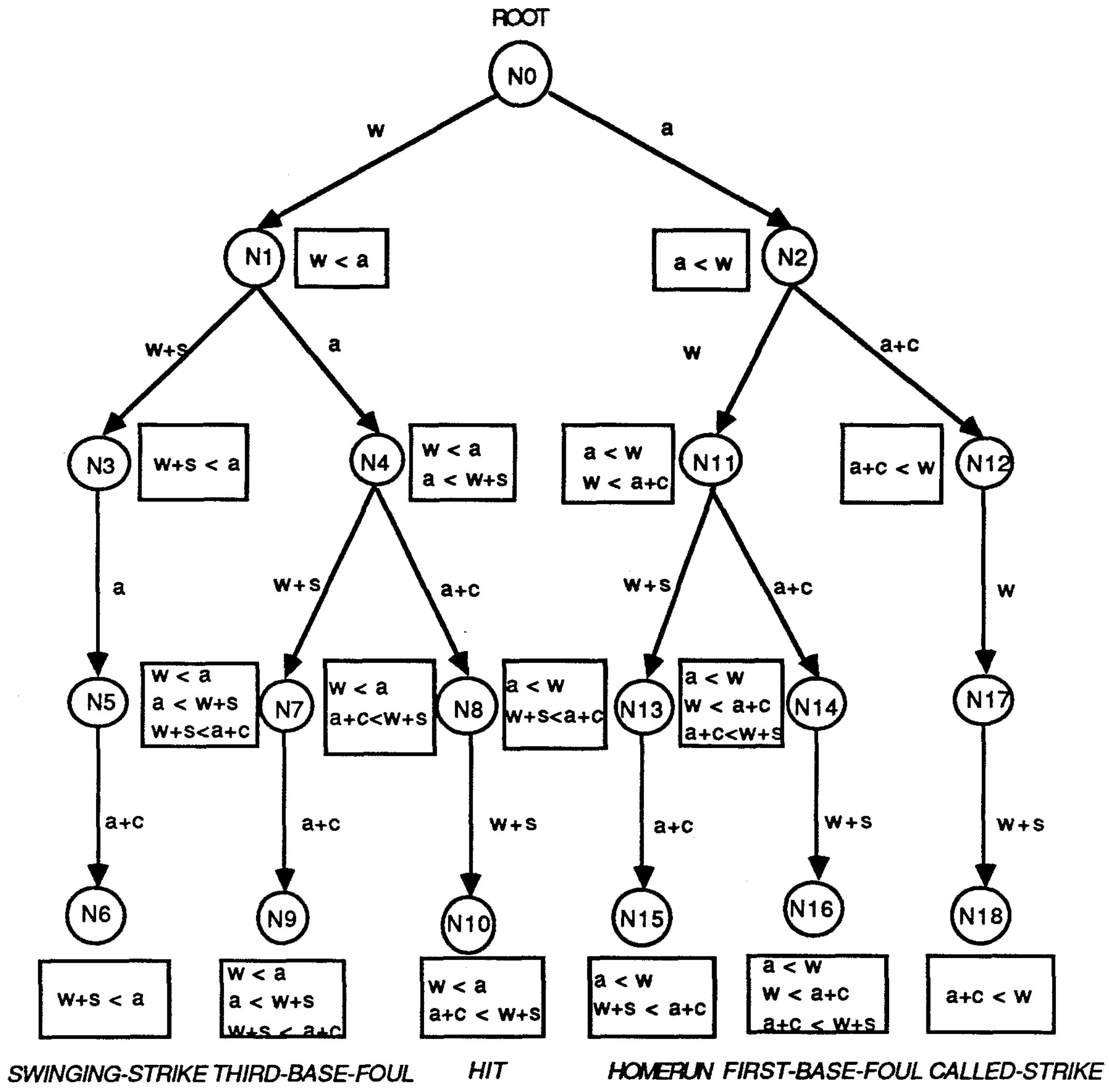


(a) Model Structure of Baseball





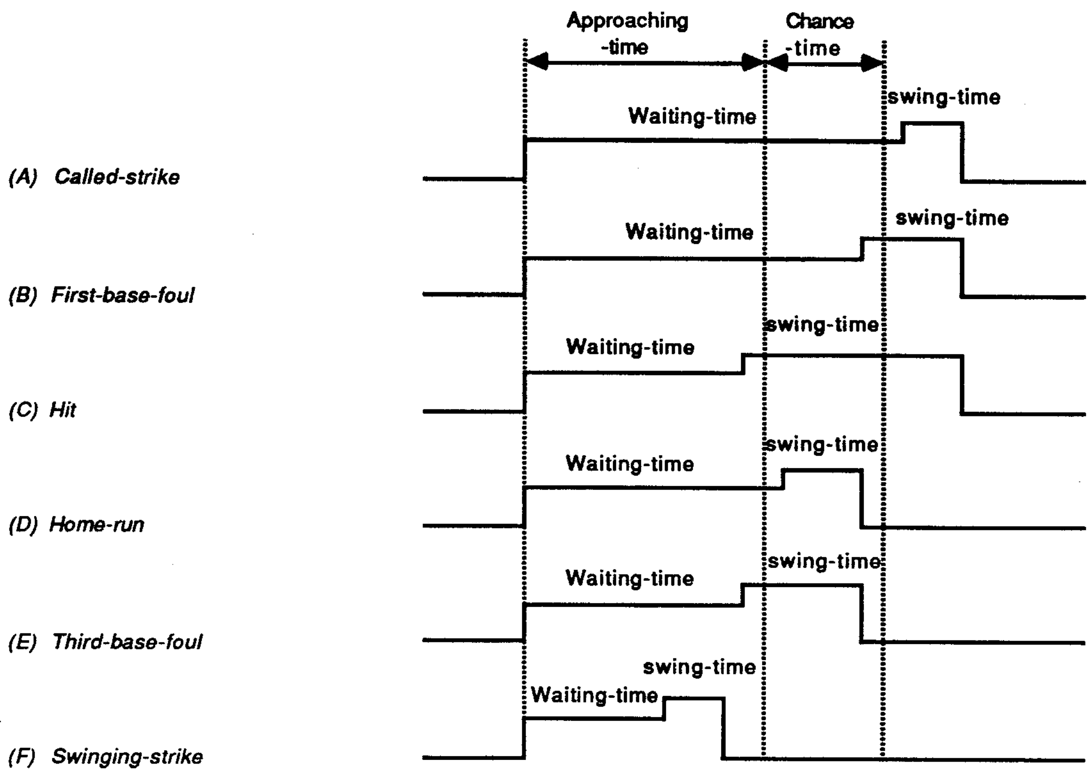
(b) State Transition Graphs



Where a : approaching-time, w : waiting-time, c : chance-time,
s : swing-time

Note : Branch is labelled by the time chosen to be set minimum
and constraints of each node are shown in an associated box.

(c) Generated Tree of Trajectories



(d) Timing Relations

Figure 4.1. Symbolic Simulation Example : Baseball

4.3. Linear Polynomial Constraint Inferencing Algorithm

4.3.1. Introduction

We have developed a linear polynomial constraint management algorithm, which is called *Reduce* in our symbolic DEVS, for inferencing the constraint sets since symbolic times are linear polynomial expressions. Such a constraint managing system, as shown in Figure 4.2, should not only accept equalities/inequalities but also check for consistency of its constraint sets. It should output the best solution possible with the available data.

Thus, as illustrated in Figure 4.3, it should have the capability to reason about consistency with new data by means of constraint propagation:

• if the new data item is consistent and not redundant, add a new constraint and delete any redundant constraint.

• if the new data item is inconsistent or redundant, the constraint set is not changed. If the new data item is inconsistent, this fact is reported to the user.

Queries should be entered in the form of new data items. If a query is consistent and redundant, then it can be derived from the current knowledge base and the answer is "yes." If it is consistent but not redundant, then the answer is "possible." If it is inconsistent, the answer is "no."

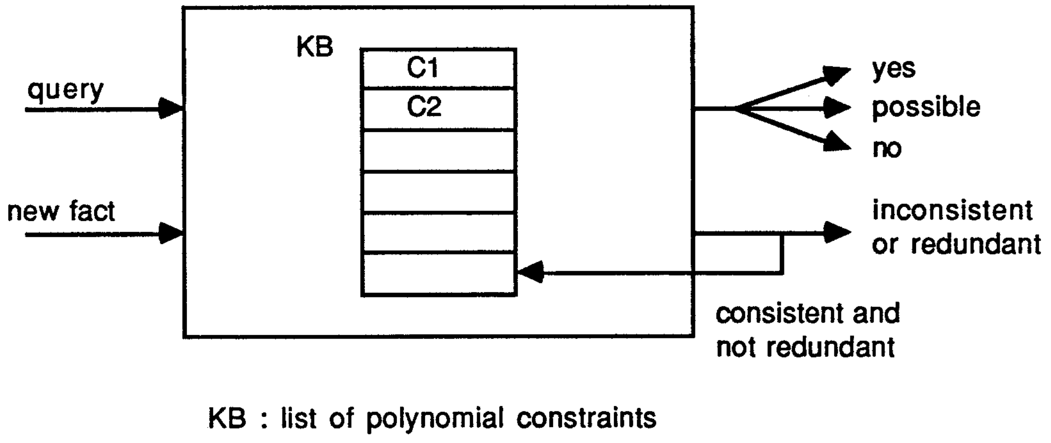


Figure 4.2. Knowledge Base System Concept

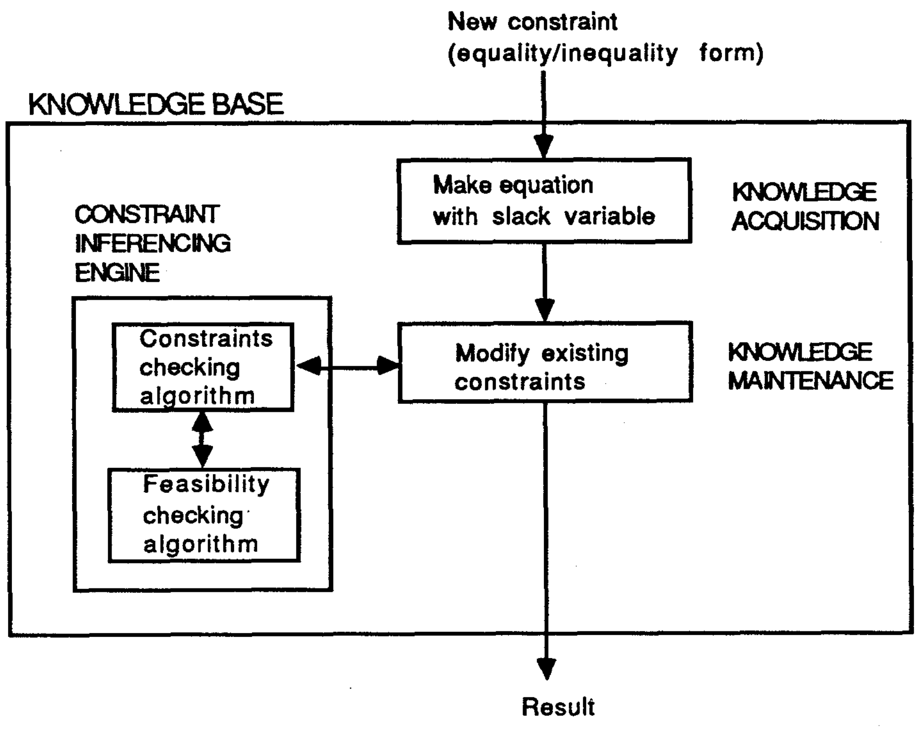

New constraint
(equality/inequality form)
KNOWLEDGE BASE
Make equation with slack variable
KNOWLEDGE ACQUISITION
CONSTRAINT INFERENCING ENGINE
Constraints checking algorithm
Feasibility checking algorithm
Modify existing constraints
KNOWLEDGE MAINTENANCE
Result

Figure 4.3. Knowledge Base System Structure

For example, we can ask our system "Was Raphael born before 1000 ?", "Did the Thirty Year War end after 1500 ?", or "Is the ratio of the width of the house to the width of the table between 2.5 and 3.5 ?". We can also update the knowledge base with new facts such as "The starting date of the Thirty Year War was between 1500 and 1700", or "The length of the house is 5 [m]". [22],[23] All the above constraints can be represented as linear polynomials by introducing slack variables as follows:

Q1 : date_birth_Raphael + slack0 = 1000

Q2 : end_date_30_year_war - slack1 = 1500

Q3 : 2.5 * width_of_table + slack2 = width_of_house

width_of_house + slack3 = 3.5 * width_of_table

C1 : start_date_30_year_war - slack4 = 1500 start_date_30_year_war + slack5 = 1700

C2 : length_house = 5

We have developed a consistency checking algorithm for linear polynomial constraints based on a feasibility checking algorithm borrowed from linear programming [67].

4.3.2. Consistency Checking Algorithm

Through an analysis of a few examples that are simple enough to see "what is going on," it will be possible to bring out certain general features of consistency problems. As illustrated in Figure 4.4, consider a consistent set of constraint $\{C_1, C_2, C_3, C_4\}$ such that:

$C_1 : x_1 \geq 10$

$C_2 : x_2 \geq 5$

$C_3 : x_1 + x_2 \leq 20$

$C_4 : -x_1 + 4x_2 \leq 20$

where $x_1, x_2 \geq 0$

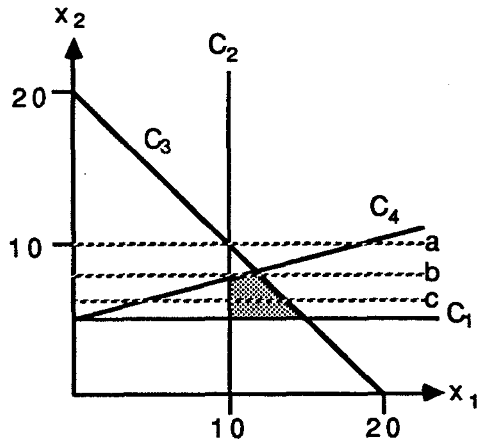


Figure 4.4. An Example

The dotted area in Figure 4.4 represents the feasible solution region. Now, let's add new constraints, *a, a'*, *b*, and *c* , respectively, such that;

$a : x_2 \geq 10$

$a' : x_2 \leq 10$

$b : x_2 \leq 7.5$

$c : x_2 \leq 7$

Depending on the constraint to be added, there exist four different cases;

• Case 1. inconsistent (example: constraint *a*) -- knowledge base unchanged, error reported.

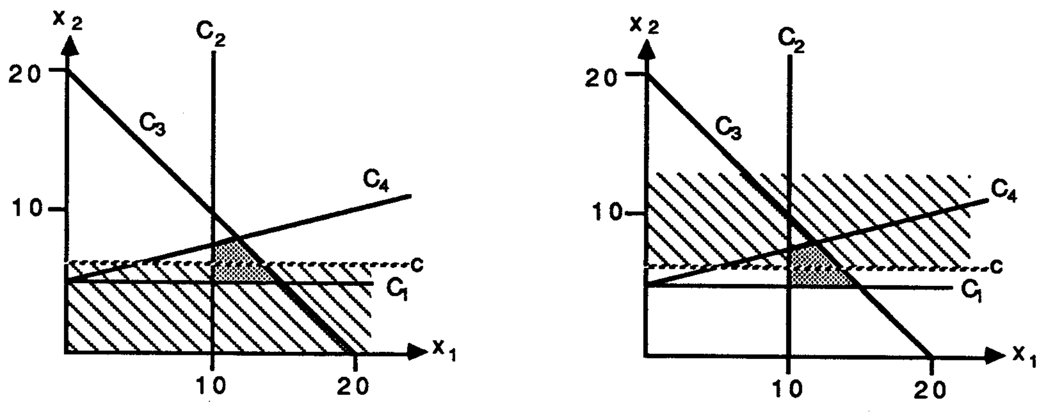
• Case 2. redundant (example: constraint *a'*) -- knowledge base unchanged.

• Case 3. consistent (example: constraint *b*) -- new constraint is added to the knowledge base.

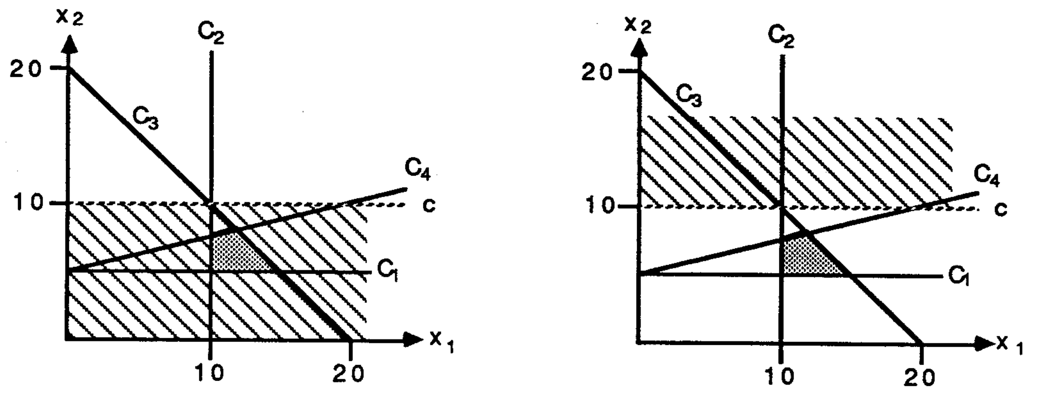
• Case 4. consistent (example: constraint *c*) -- new constraint is added to the knowledge base, but simultaneously, the now redundant previous constraint, C_4, is removed from the knowledge base.

Inconsistency can be checked easily: if the new constraint is infeasible, then it is inconsistent, otherwise one of the cases 2, 3, or 4 applies.

Redundancy can be detected by considering the negation of the constraint. If the new constraint is not redundant but consistent, then the negated constraint is also not redundant but consistent (see Figure 4.5a). However, if the new constraint is redundant, then the negated constraint is inconsistent (see Figure 4.5b). It can therefore be discriminated using the previously discussed method.



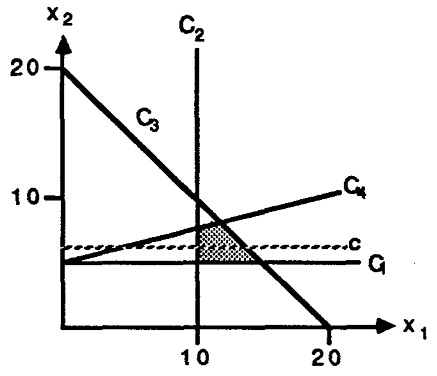
(a) Irredundant Case



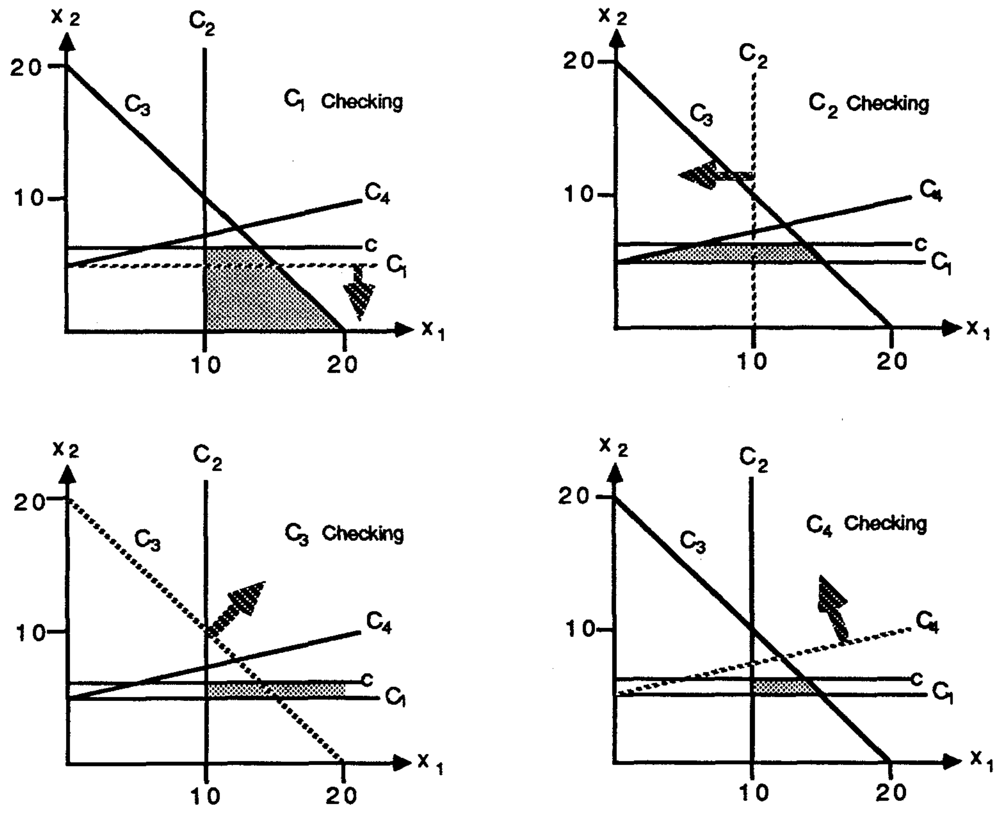
(b) Redundant Case

Figure 4.5. Redundancy Checking

Once a new constraint has been shown to be consistent yet not redundant, we should remove all constraints from the knowledge base that have meanwhile become redundant. As shown in Figure 4.6, suppose the new constraint, *c* ($x_2 \leq 7$), is added to the knowledge base, then C_4 should be eliminated from the knowledge base since it has become redundant. Thus, we have to test each existing constraint by looking at its negation. In Figure 4.6, the C_1, C_2, and C_3 constraints are irredundant since $\neg C_1$, $\neg C_2$, and $\neg C_3$ are consistent, but C_4 has become redundant since $\neg C_4$ is inconsistent. Figure 4.7 illustrates the overall algorithm for constraint checking, so that only feasibility checking is left to be discussed.



(a) Constraint *c* is added



(b) Iterating Redundant Constraint Checking

Figure 4.6. Redundant Constraint Elimination

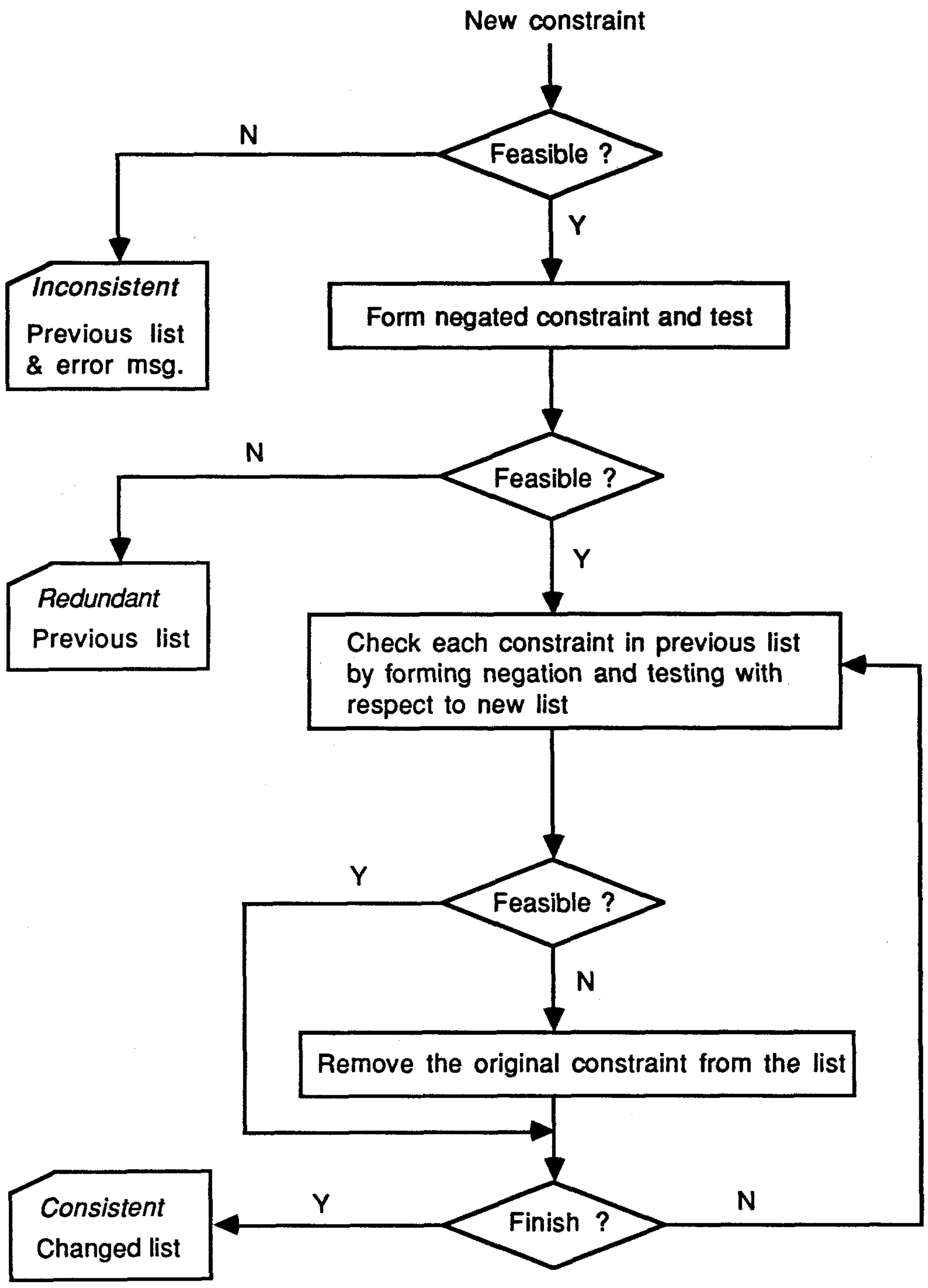

New constraint
Feasible ?
N
Y
Inconsistent
Previous list
& error msg.
Form negated constraint and test
Feasible ?
N
Y
Redundant
Previous list
Check each constraint in previous list by forming negation and testing with respect to new list
Feasible ?
Y
N
Remove the original constraint from the list
Consistent
Changed list
Y
Finish ?
N

Figure 4.7. Consistency Checking Algorithm

4.3.3. Feasibility Checking Algorithm

A pivotal algebra algorithm introduced by Orden efficiently replaces Phase I of the simplex method without reference to artificial variables or an artificial objective function[67]. Most linear programming methods deal with closed boundary relations only such as $\leq$, $=$, and $\geq$, not with open boundary relations such as $<$ and $>$ in that most techniques are based on boundary checking (minimax problem). However, we must be able to handle open boundary relations due to our desire to evaluate negated constraints. For example, the negation of the constraint $x \leq a$ is $x > a$. However, by introducing a new slack variable ε, the open boundary relation can be replaced by a closed boundary relation preserving mathematical equivalence. For example, if "statement A is true for all $x < a$", we can equivalently say that "statement A is true for all $x \leq a - \varepsilon$ where $\varepsilon > 0$".

<u>Brief description of pivotal algebra algorithm</u>

The general linear equality/inequality system can be stated as follows:

Find a vector $[x_1, \ldots., x_n]$ that satisfies

$$\sum_j a_{ij} x_j = b_i \quad i = 1, \ldots, m$$

$$x_j \geq 0, \quad j = 1, \ldots, n$$

Note that inequalities can always be converted to equalities using slack variables. The right hand side (b_i) can easily be made non-negative by multiplying both sides of the equality by -1.

The following definitions will be used:

* *"reduced"* : The i-th row of matrix A is called "reduced," if it contains at least one 1 element, e.g. in column j, and if all other elements of the j-th column of matrix A

are zero, i.e., if the j-th column of matrix A consists of the i-th unit vector, or if the i-th row contains only 0 elements.

* [*A b*] : is called the ***augmented matrix*** of the linear system Ax = b.

* ***"homogeneous"*** : The i-th row of the augmented matrix [A b] is called "homogeneous," if the b_i element is 0.

* ***"semi-reduced"*** : The augmented matrix [A b] is called "semi-reduced" if every row is either reduced or homogeneous.

If a system can be made semi-reduced, then it has a feasible solution, otherwise it has an infeasible solution.

Pivot procedure:

As shown in Figure 4.8, choose a non-reduced row, i_D. If $a_{i_Dj} \leq 0$, $j = 1, \ldots, n$, the system is infeasible. Otherwise, find the column and row:

$j_P = j \mid \underset{j=1,..,n}{\text{Max}}\ a_{i_Dj}$ where j_P is the index of the pivot column.

$i_P = i \mid \underset{a_{ij_P} > 0}{\text{Min}}\ b_i/a_{ij_P}$ where i_P is the index of the pivot row.

Then pivot at (i_P, j_P). There are four cases to consider.

• Case 1. ($i_P = i_D$) : the driving row is the pivot row. In this case, it becomes a reduced row.

• Case 2. ($i_P \neq i_D$, and at least one $a'_{i_Dj} > 0$) : The driving row cannot be reduced. In this case, repeat the pivot procedure with the driving row unchanged, until it becomes a reduced row (Case 1), or until either Case 3 or Case 4 applies.

• Case 3. ($i_P \neq i_D$, all $a'_{i_Dj} \leq 0$, and $b'_{i_D} = 0$) : proceed to another driving row.

• Case 4. ($i_P \neq i_D$, all $a'_{iDj} \leq 0$, and $b'_{iD} > 0$) : The system is infeasible.

Orden [67] proved the convergence of the algorithm. Its complexity is of order $0(nm^2)$ since at most m rows must be semi-reduced, each requiring at most n positions to pivot, each pivot requiring at most m operations. This deterministic polynomial complexity contrasts with the non-deterministic polynomial complexity required for testing satisfiability of first order logic [32].

4.3.4. Relation to "Commonsense" Arithmetic Reasoning

Simmons has developed the so-called *Quantity Lattice* to perform a wide range of "common sense" arithmetic inference operations [83],[84]. The *Quantity Lattice* integrates qualitative and quantitative reasoning and combines inequality reasoning with reasoning about simple arithmetic expressions. It uses five reasoning techniques of i) graph search, ii) numeric constraint propagation, iii) interval arithmetic, iv) relational arithmetic and v) constant elimination arithmetic. Simmons also outlined the advantages of the *Quantity Lattice* by comparing it with other approaches such as the temporal reasoning approach of Allen [5], and the DEVISER planning system by Vere [88].

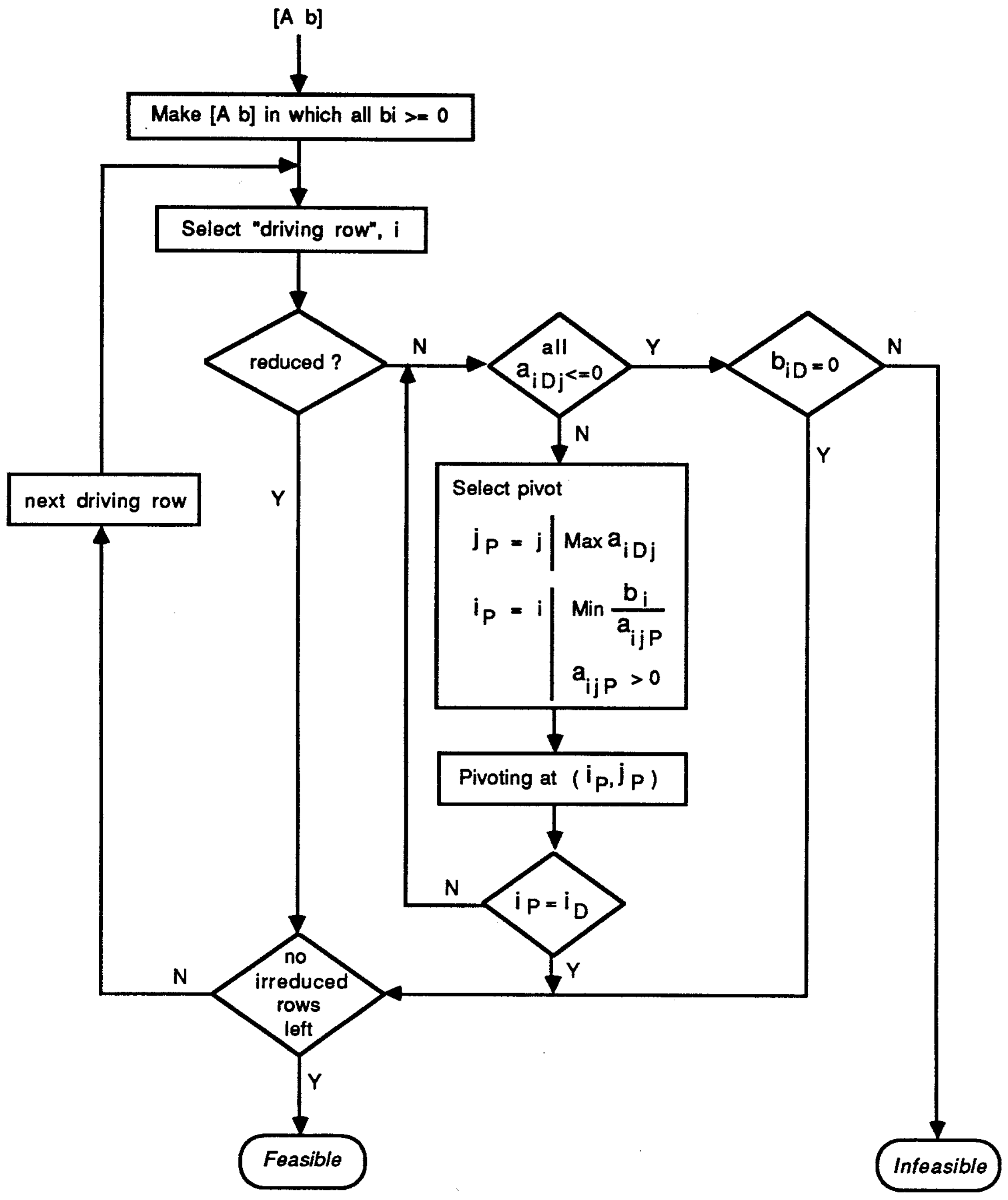


Figure 4.8. Feasibility Checking Algorithm

In contrast with Simmons' work, our algorithm for linear polynomial constraint management using linear programming techniques has been developed for the purpose of symbolic simulation in which the simulation time and the branching constraints due to non-determinism are represented by linear polynomials. The proposed algorithm can efficiently and coherently integrate various reasoning approaches of Simmons for constraints represented by linear polynomial constraints with $>$, $<$, $=$, $\geq$, and $\leq$ relationships.

In the following, we shall briefly describe the five arithmetic reasoning techniques and corresponding examples introduced by Simmons, and show how they can be handled using the linear polynomial representation technique employed in our algorithm.

i) Graph Search: Graph search is a qualitative technique to determine the relationship between two quantities. This technique searches the graph of quantities using a simple breadth-first search algorithm to find a path between the investigated quantities.
Example : if $A = C$, $C < E$ then $A < E$?
Linear Polynomial Representation:
if $A - C = 0$, $C - E + slack1 = 0$, then $A - E + slack2 = 0$?

ii) Numeric Constraint Propagation: This is a quantitative technique to determine the relationship between two quantities. The ordering between two quantities can be determined if the intervals associated with the quantities do not overlap, except possibly at their endpoints, since the value of a quantity is constrained to lie within the interval.
Example : if $A < 1$, $B > 2$, then $A < B$?
Linear Polynomial Representation :
if $A + slack1 = 1$, $B - slack2 = 2$, then $A - B + slack3 = 0$?

iii) Interval Arithmetic: This technique computes the value of an arithmetic expression by applying the arithmetic operator of the formula to the endpoints of intervals of its arguments.

Example : if $3 < A < 4$, $1 < B < 4$, then $4 < A + B < 8$?

Linear Polynomial Representation :

if $A - slack1 = 3$, $A + slack2 = 4$, $B - slack3 = 1$, $B + slack4 = 4$,

then $A + B - slack5 = 4$, $A + B + slack6 = 8$?

iv) Relational Arithmetic: This arithmetic maintains constraints on the qualitative relationship of an arithmetic expression to its arguments. The relationship depends on the relationship of the expression or its arguments to the identity value for the arithmetic operator of the expression.

Example : if $X = Y + 5$, then $X > Y$?

Linear Polynomial Representation :

if $X - Y = 5$, then $X - Y - slack1 = 0$?

v) Constraint Elimination Arithmetic: If the same amount is added to two expressions then the results are related in the same way as the original expressions are.

Example : if $A = B + X$, $C = D + X$, $B > D$, then $A > C$?

Linear Polynomial Representation :

if $A - B - X = 0$, $C - D - X = 0$, $B - D - slack1 = 0$,

then $A - C - slack2 = 0$?

The developed quantity knowledge base system coherently maintains the mathematical relations in the form of linear polynomial constraints on quantity variables such as times and duration, etc. and performs inferences on them. The inference mechanism for handling linear equality/inequality constraints called consistency checking algorithm based on the feasibility checking algorithm borrowed from linear programming has been successfully developed and tested. In contrast with other approaches such as the quantity lattice approach due to Simmons, our algorithm can efficiently and coherently integrate various reasoning approaches represented by linear polynomial forms with $>$, $<$, $=$, $\geq$, and $\leq$ relationships.

4.4. Conclusions

An extended discrete event specification formalisms, symbolic DEVS, has been defined that extends the time base from the real numbers to the linear polynomials over the real numbers. This permits next event times to be expressed symbolically, although precise knowledge of any event time can be readily incorporated by setting its symbol to a real value. Therefore symbolic DEVS is truly an extension of conventional DEVS and reduces to the latter when all event times are real valued. A network of symbolic DEVS components is non-deterministic since the minimum of a set of next event times is not unique as in a conventional DEVS of the same kind. A simulation algorithm was developed to generate the branching trajectories resulting from the underlying non-determinism. The extended formalism offers a convenient means to conduct multiple, simultaneous explorations of model behaviors. Since symbolic times are linear expressions, techniques from linear programming were adapted to efficiently manage and inference the constraint sets [67].

We have shown how symbolic DEVS can perform the causal propagation required for diagnosis of fault-prone systems modelled with multicomponent DEVS. By allowing the event times to be expressed symbolically, we generate a family of trajectories that represent the possible sequences of events resulting in a particular breakdown. Such a family can be employed to trace a detected anomaly back to the components that might have caused it. The approach requires that we represent all failures that could occur and how systems respond to them. In Chapter 9, we will show how our simulation environment helps to visualize and discover, via actual execution, anomalies that need to be incorporated as faults. Guided by morphism preservation principles, we intend to develop tools to facilitate incorporating new fault types into model units directly from observations of simulation runs.

CHAPTER 5
ROBOT-MANAGED LABORATORY OF THE FUTURE

5.1. Overview

As a concrete example of a high autonomy system, let us consider a space-borne fluid handling laboratory environment to be operated by robot organizations. This generic laboratory environment may serve as a prerunner of any chemical, or similar, laboratory to be realized aboard the forthcoming International Space Station Freedom. Laboratory management includes servicing and calibration of equipment, set-up of experiments in progress, measurement and recording of data, and finally analyzing of data. However, it also includes monitoring of laboratory performance, detection and discrimination of faults, recovery from fault situations, and adjustments of operational plans as a consequence of unavailable resources due to previously discovered and still uncorrected faults.

The environment should :

- support the development of robot cognitive systems and strategies for effective multi-robot management of laboratory experiments,
- support a study of the trade-off in partitioning tasks between hard and soft forms of automation, i.e., we could have a robot-free laboratory containing a large number of special purpose "intelligent" instruments capable of almost complete self-control or a small number of very flexible robots that can be trained to manage a small number of general purpose instruments,
- support various degrees of human-robot cooperation, allowing many tasks to be performed by humans initially that can later be taken over by robots as the technology for such automation becomes available,
- support the development of workable human-robot cooperation protocols, and
- allow the assessment (through simulation) of the difficulties and risks resulting from humans and robots sharing the same workspace [109].

Time has come to investigate advanced robot-controlled instrumentation schemes. For example, handling fluids in orbit will be essential to many of the experiments being planned in space manufacturing and biotechnology. However, the microgravity conditions of space necessitate radically different approaches to fluid handling than common on earth. As experience in space accumulates, approaches and instrumentation used in space will likely undergo continual modification, enhancement, and replacement. Therefore, instruments used aboard one and the same spacecraft may vary over time, while the robots operating on these instruments will remain the same. Simultaneously, it may not always be possible to repair faulty equipment quickly (e.g. on planet Mars) due to the difficulty to access the equipment from Earth. Thus, robots for managing such equipment must be highly autonomous and flexible so that they can operate efficiently and effectively in an environment that is constantly changing due to system degradation and upgrading.

The project is to develop a technology that will allow carefully designed and specially constructed laboratory robots to perform many routine tasks in a space laboratory under remote supervision from the ground as shown in Figure 5.1. Therefore the robots must be able to perform simple operations autonomously, and communicate with the ground only at the task level and above [4]. Such robots should be able to judge the adequacy of a proposed action plan on the basis of expectations of its effects on the laboratory, materials, instruments, etc. For this purpose, it is important that simulation models at various levels of granularity can be automatically generated at run time from a set of generic master models [10],[11].

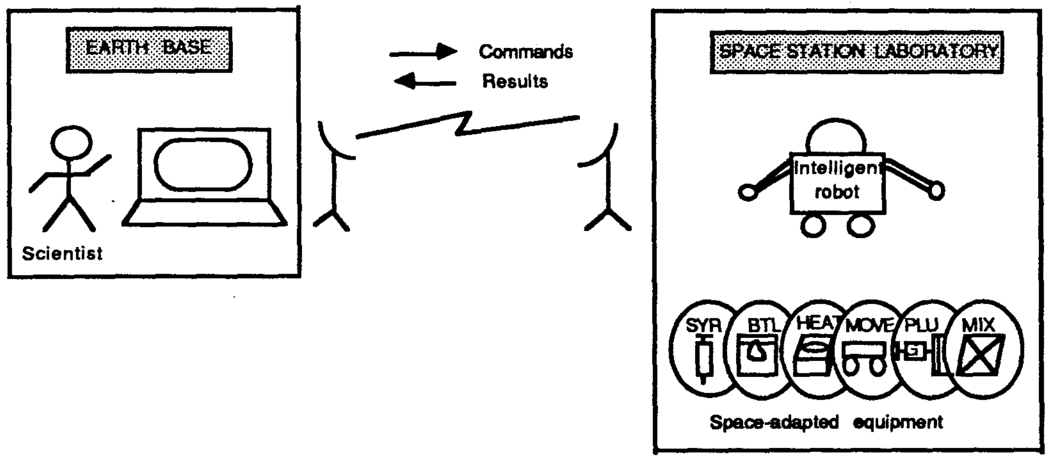


Figure 5.1. Tele-autonomous System Concept

In designing the robot models, we assume that necessary mobility, manipulative and sensory capabilities exist so that we can focus on task-related cognitive requirements. Such capacities, the focus of much current robot research, are treated at a high level of abstraction obviating the need to solve current technological problems.

In this chapter, we briefly illustrate an overall domain knowledge representation by using the system entity structure to build a high autonomy system through the model-based design of a robot-managed laboratory.

5.2. SES Representation of a Space Laboratory

The laboratory environment illustrated in Figure 5.2 is realized on the basis of object-oriented and hierarchical models of laboratory components within DEVS-Scheme. Laboratory configurations will be determined by issuing commands that invoke a pruning operation of the entity structure knowledge representation. The laboratory model is designed to be as generic as possible. However, as stated earlier, the focus will be upon fluid handling in microgravity, which presents a variety of problems that are unique to space.

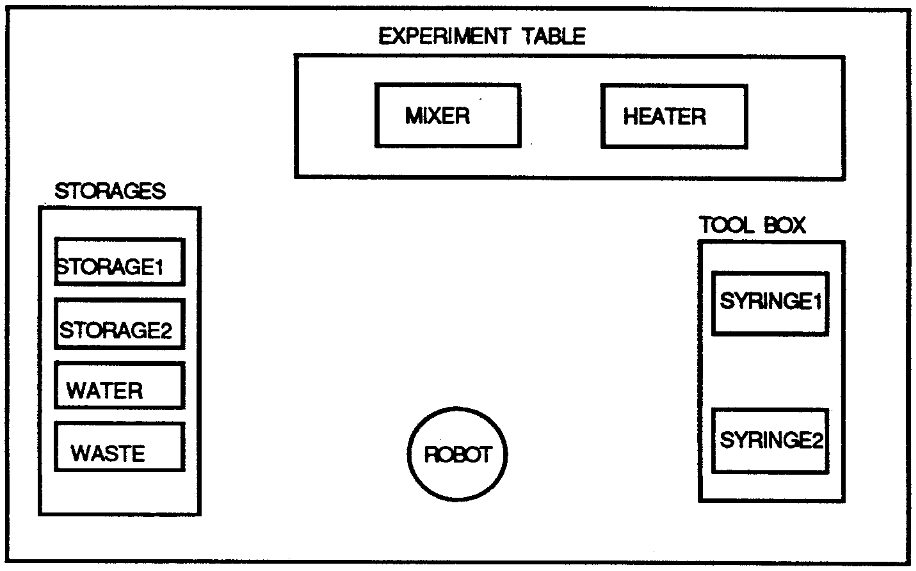


Figure 5.2. Space-borne Chemical Laboratory View

Figure 5.3. illustrates the approach taken. The entity structure for the Space Station Laboratory (SSL) decomposes this entity into its structure knowledge part and its experiment (goal) knowledge part; STRUCTURE and EXPERIMENT. The former relates to the execution structure which is concerned with how to construct the system, while the latter relates to the goal structure which is concerned with how to achieve a desired goal efficiently (details are discussed in Chapter 9). After pruning the STRUCTURE, the entity structure is transformed into a hierarchical model containing *controlled models* at two levels. Each of the entities will have one or more classes of objects (models) expressed in DEVS-Scheme to realize it.

The EXPERIMENT contains a three level abstraction hierarchy distinguishing between high-level task planning models (HM), middle-level task planning models (MM), and low-level task planning models (LM) that are associated with so-called model plan units (MPUs): HMPU, MMPU, and LMPU in the STRUCTURE part, respectively, which are the cognitive control elements of the system. Each level is designed to represent experimental knowledge needed to decompose a given task into a composition of subtasks.

The SPACE and OBJECTS decomposed from STRUCTURE are designed for the simulation-oriented knowledge representation. STRUCTURE is implemented as a *controlled model* within the DEVS environment. The SPACE is the *controller*, whereas the OBJECTS are the *controllees* of the SSL controlled model. The SPACE is basically a modelling artifact -- necessitated by the object-oriented, modular modelling paradigm -- to conveniently represent knowledge of where objects are and with whom they can communicate and interact. Motion and communication of ROBOTs are managed by the SPACE. When a ROBOT moves around, its MOTION system sends its new location and direction to the SPACE which keeps track of the ROBOT's positions and directions. When a ROBOT wishes to communicate with other ROBOTs, it sends message via its SENSE system to the SPACE which relays the message only to those ROBOTs within the range of the sender. The range is determined by the channel on which the message is sent. Thus different transmission media and sensory modalities can be modeled, such as light and vision, sound and hearing, pressure and touch, etc.

Since the SPACE has complete knowledge of locations, it can detect collisions between ROBOTs. SPACE is viewed as a kind of resource shared by its occupants so that collisions represent attempts to occupy the same space more than once at the same time. The SPACE can report such an event but do nothing to prevent it. However, the SPACE may be given greater intelligence to co-ordinate the ROBOTs, for example to prevent collisions, and to perform other space resource management tasks. In this case, it assumes the role of an "actual" system component, at least in part.

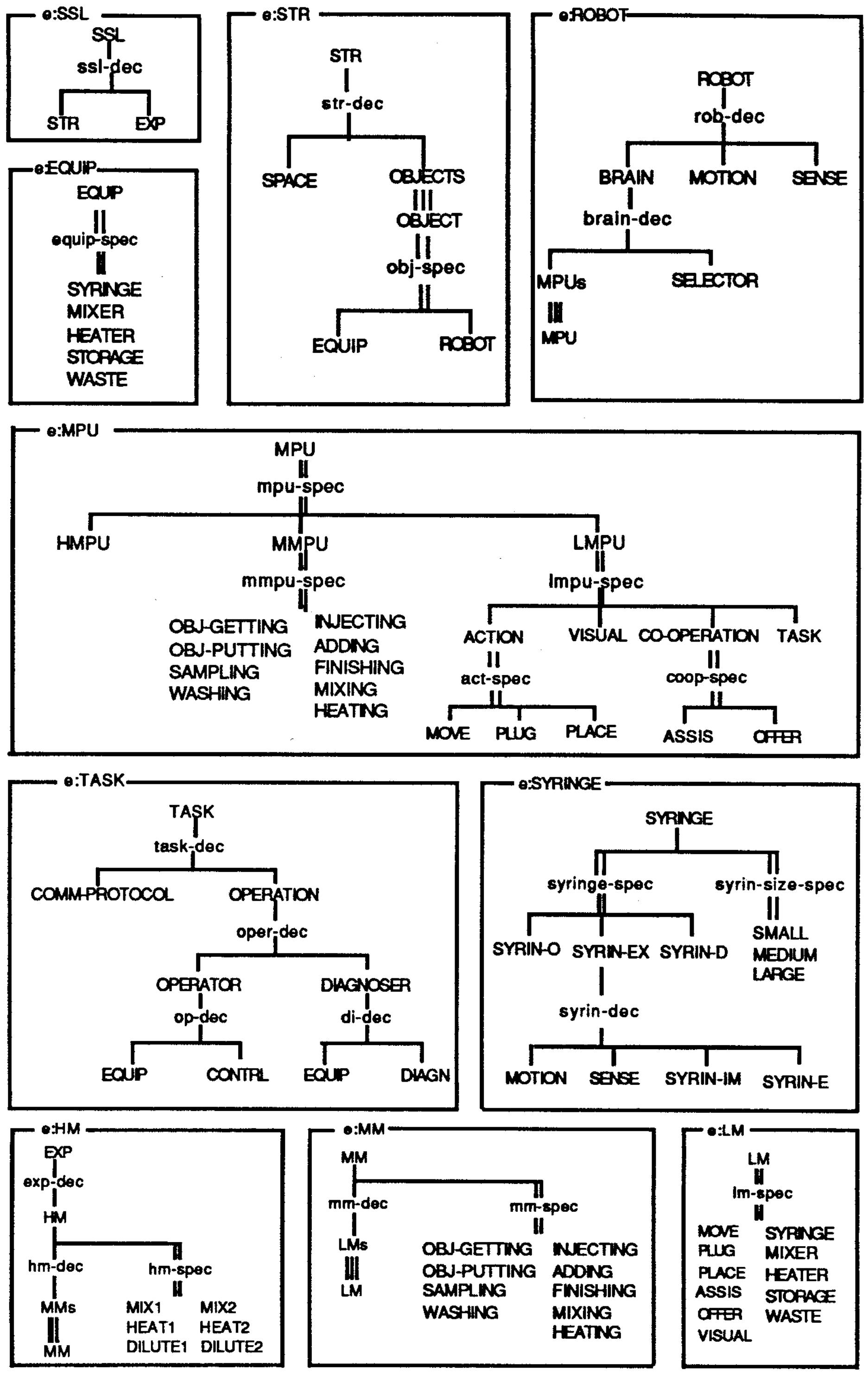


Figure 5.3. Partitioned SES of Robot-Managed Fluid Handling Laboratory

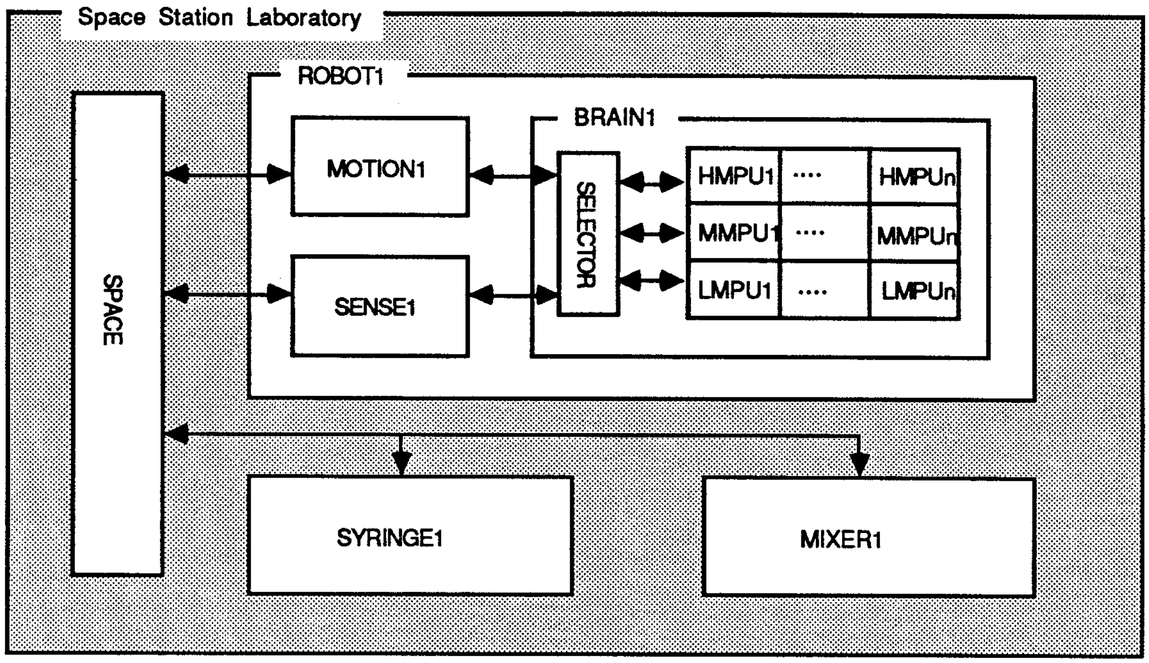


Figure 5.4. Robot System Organization Example

Each OBJECT is specialized into ROBOT and EQUIP. Each ROBOT is decomposed further into MOTION, SENSE, and BRAIN. The EQUIP is a generic entity for the laboratory equipment that is modeled much in the same way as the ROBOT. However, EQUIP has no BRAIN, and its MOTION and SENSE subsystems are always passive. Note that OBJECTS are also defined as a *multiple entity*. Therefore, any number of OBJECTS may be generated, and with the pruning discussed earlier, we can have any desired number of ROBOTs and EQUIPs in the laboratory.

Each Robot's Cognition-system (BRAIN) is also implemented as a *controlled model* containing a SELECTOR as its controller, and several MPUs as its controllees.

The SELECTOR provides the communication channel between the SENSE subsystem and the MPUs. Essentially it is a bi-state (open/closed) device whose state is determined by the MPU responses. In its closed state, it passes on the incoming sensory inputs to the activated MPU. Upon completion of the activated plan or upon receiving a discrepancy alert, it switches to the open state in which MPUs may vie for activation.

The MPUs are specialized into HMPUs, MMPUs, and LMPUs, respectively, corresponding to the EXPERIMENT abstraction hierarchy previously mentioned. HMPU is a high level MPU that manages the actions of the middle level MMPUs, each of which performs the same function with respect to the low level LMPUs. The latter employs event-based control logic to interact with the real world. These three types of MPUs are represented at the same level in the entity structure but conceptually they reflect the hierarchical decomposition structure of the EXPERIMENT models.

The LMPUs comprising the robot's brain are of two kinds: those specialized for carrying out specific tasks, and those specialized for more general tasks involving communication, motion, vision, cooperation, etc.

- TASK_LMPU: is a LMPU specialized for executing a particular task (i.e., is further specialized in the SES). When help or visual identification of an object is needed in performing this task, it relinquishes control to the ASSIS_LMPU or VISUAL_LMPU, respectively. A detailed discussion of the decomposition of the TASK_LMPU is given in Chapter 7.
- ASSIS_LMPU: is a LMPU specialized for the task of requesting help from other robots. When it is activated, it initiates a COMM_PROTOCOL which tries to make contact with ROBOTs within its range and engage one that can provide the needed assistance.
- OFFER_LMPU: is a LMPU specialized for the task of dealing with incoming requests for help emitted from ASSIS_LMPUs of other ROBOTs. When activated, it decides if help can be offered, and if so, engages in a dialogue with the ASSIS_LMPU of the help-seeking ROBOT and sets up a rendez-vous. It relinquishes control to the ACTION_LMPU to bring the ROBOT to the requestor's work site.
- ACTION_LMPU: is a LMPU specialized for directing the motion subsystem to bring the ROBOT to a given destination or position. It requests the current motion state from the MOTION component, and sends it new parameters (direction,

speed, and time-step) for traveling to the vicinity of the destination. Once there, it directs the MOTION component in physically contacting the object or ROBOT at the destination. The touch channel is used for judging when contact has been established. Three ACTION_LMPUs are being implemented so far; MOVE_ACTION_LMPU for robot walking, PLUG_ACTION_LMPU for pull/push motion, and PLACE_ACTION_LMPU for put/get motion.

- VISUAL_LMPU: is a LMPU specialized for the task of visual identification of objects. It relinquishes control to ACTION_LMPU to bring the ROBOT to new locations when different viewing distances and perspectives are needed. Once there, it resumes control to accept a visual image and to identify objects by consulting its built-in recognition system.

This concludes the description of the structural and experimental decomposition of the space-borne fluid handling laboratory environment as described by its SES. This decomposition will be frequently referred to in the subsequent chapters, which also fill in the details of how the structural decomposition is actually being used for various purposes such as task planning, intelligent control, and fault diagnosis.

CHAPTER 6
MODEL-BASED CONCEPTS FOR HIGH AUTONOMY SYSTEMS

6.1. Overview

Autonomy is an extended paradigm that subsumes both control and AI paradigms, each of which is limited by its own abstractions. Control theory has run up against the limitations of its rigorous, but sparsely applicable, mathematical framework. AI research has lived under the illusion that intelligence can be demonstrated in abstracted symbol spaces bereft of a rich, continuous coupling to the real world. Autonomy, as a design goal, offers an arena where both control and AI paradigms can be applied -- and a challenge to the viability of both as independent entities.

In the last seven years, an approach called *intelligent control* has begun to take root. Autonomous control systems must perform well under significant uncertainties in the plant and environment for extended periods of time and they must be able to compensate for system failures without external intervention. Advanced planning, learning, and expert systems, among others, must work together with conventional control systems in order to achieve autonomy. It is clear that the development of autonomous controllers requires significant interdisciplinary research efforts as it integrates concepts and methods from areas such as control, identification, estimation, and communication theories, computer science, especially AI, and operations research.

Researchers concerned with enhancing robotics beyond pure muscle amplification are arriving at similar conclusions as in Figure 6.1. [45].

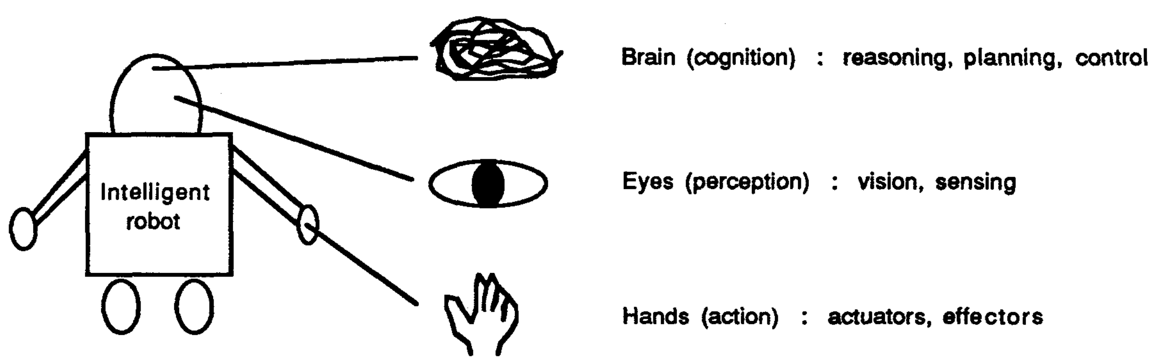

Intelligent robot
Brain (cognition) : reasoning, planning, control
Eyes (perception) : vision, sensing
Hands (action) : actuators, effectors

Figure 6.1. Components of Autonomy

From a robotics perspective, robotics enables the realization of autonomous systems, which include three basic components. One is to perceive, understand, and sense inputs from the outside world. The second relates to intelligent, sensible, and meaningful decision making. The last is concerned with action; it uses manipulation to have an effect on the outside world. AI techniques are involved in realizing the middle component.

Naturally, we can expect AI researchers to see things differently. Indeed, Minsky sees the knowledge-based approach adopted by AI as a generalization of the feedback control paradigm. Although the AI approach is much more crude than classical feedback control, it can nevertheless work. However, actual research in AI has, since its inception, not concerned itself with the problems of living in the real world. It has limited the environment with which its systems must cope to abstract problem spaces such as chess, word matching, and concept learning. Thus, really interesting problems are going to involve the integration of symbolic problem-solving with perception in a robotics environment. Intelligence must not be treated as a separate problem solver using only abstract knowledge representations. AI researchers have a tendency to formulate problems abstractly, and ignore the fact that for many of these problems, perception and motor control may provide useful solutions that can be integrated with planning and decision making. Not only is high autonomy best viewed as a problem of integration of the three modalities, but the AI subtask itself will, in the long run, be easier to tackle if we treat it in conjunction with the perception and action subtasks.

6.2. Architecture for High Autonomy Systems

To include such diverse systems as planetary colonies, self-operating factories, and telerobotic laboratories along with autonomous land and space vehicles in the autonomy umbrella, we employ a definition as it is used by NASA [63]:

> *Autonomy is the ability to function as an independent unit or element over an extended period of time, performing a variety of actions necessary to achieve pre-designed objectives while responding to stimuli produced by integrally contained sensors.*

In this view, automation and robotics are technologies available to help attain autonomy.

Saridis, a pioneer in the field, developed a three layer hierarchy for intelligent control [78], which is meant to reflect increasing intelligence with decreasing precision. Antsaklis *et al.* refine the hierarchy to an arbitrary number of layers, depending on the particular application. A hierarchy with less than three levels seems not meaningful. A minimal hierarchy, as shown in Figure 6.2, contains the following three hierarchy levels (in the order of increasing authority):

1. *execution layer* : contains the systems to control actuators and effectors on the basis of sensor supplied data and control policies determined at higher levels. Such control systems include conventional automatic controllers as well as decision elements based on dynamic and symbolic models operated fast enough for real time response.
2. *coordination layer* : links the management and execution layers. It translates the high level commands generated at the management layer into sequences of actions executable at the execution layer.
3. *management layer* : is responsible for deploying the systems in the coordination layer. It thus determines the overall system goals and resolves conflicts involving

lower level goals. It also contains the human communication interface which supports knowledge engineering (transferral of expertise and other knowledge), information queries (concerning past, present or expected behavior), and supervisory commands, including overrides of lower level decisions.

Besides directives flowing from higher to lower layers, there are feedbacks flowing in the reverse direction reporting the actual system status. There is a disjoint architecture of information flow:

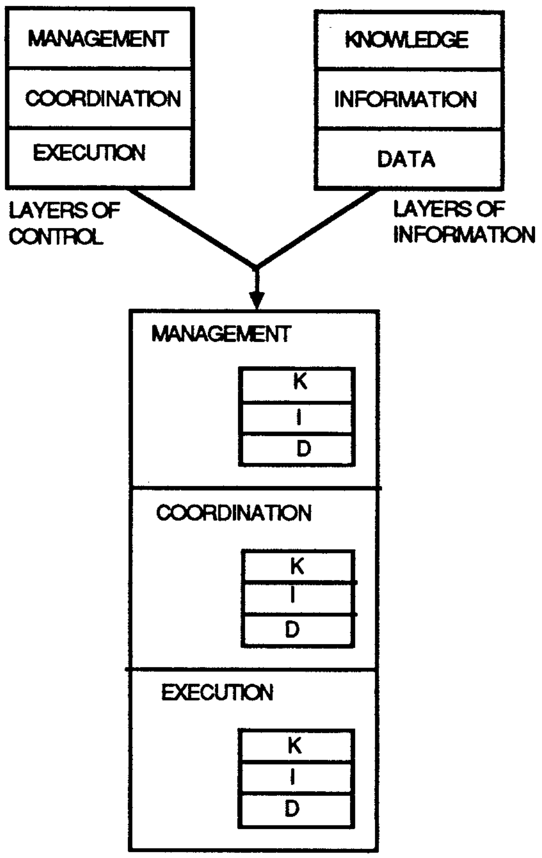


Figure 6.2. Architecture for High Autonomy

1. *data layer* : this layer contains the components that sense the system environment (both external to its "skin" as well as internal). Sensors range from direct measurement devices for environmental variables to sophisticated data transducers that reduce multivariate sensory signals into meaningful symbols.

2. *information layer* : contains data management systems that keep track of the data captured at the data layer and organize this data into a complete history of the physical and operational states of the system. This includes time stamped records of both environmental conditions and concurrent control and management events to facilitate subsequent analysis. Such behavioral series may also be compressed in the from of summary models.
3. *Knowledge layer* : contain the knowledge base for the system structure and behavior. Structural knowledge includes decomposition, taxonomic, and coupling relationships involving the physical, biological, environmental, and informational components. Behavioral knowledge is represented in simulation models. Dynamic models stored in a model base can regenerated and predict dynamic behavior of controlled processes. Symbolic models embody the know-how needed for management functions such as planning and scheduling of activities, diagnosing malfunctions, etc.

6.3. Model-Based Architecture

A simplification that has proven fruitful in our work is the model-based architecture. In this architecture (Figure 6.3), knowledge is encapsulated in the form of models that are employed at the various control layers to support predefined system objectives. As suggested above, lower layers are more likely to employ conventional differential equation models with symbolic models more prevalent at higher layers. A key requirement is the systematic development and integration of dynamic and symbolic models at different layers. In this way, traditional control theory, where it is applicable, can be interfaced with AI techniques, where they are necessary. Antsaklis *et al.* [6] propose that so-called hybrid modelling methodology must be adopted -- the combination of traditional differential/difference equation models with more recent symbolic formalisms [114]. Discrete event models, facilitating event-based control, can be employed to map dynamic to symbolic models [102],[109]. It is not totally clear yet where, in the design of a high autonomy system, DEVS and other non-traditional formalisms, such as qualitative models

[53],[91] and neural networks, play their most important role, how they are best exploited individually, and how they can be integrated into a coherent whole.

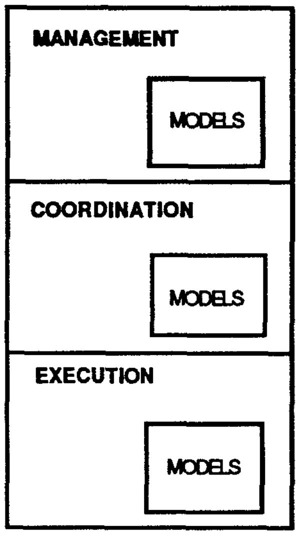


Figure 6.3. Model-based Architecture

Note that an autonomous system could in principle, base its operation, diagnosis, repair, planning, and other activities on a single comprehensive model of its environment. However, such a model would be extremely unwieldy to develop and lead to intractable computations in practice [108]. Instead, our architecture employs a multiplicity of partial models to support system objectives. As indicated, such models differ in level of abstraction and in formalism. The partial models, being oriented to specific objectives, should be both easier to develop and better computationally tractable [108]. However, this approach leads to sets of overlapping and redundant representations. Concepts and tools are needed to organize such representations into a coherent whole. The SES/MB environment [108],[109],[115] can connect models at different levels of abstraction so that they can be consistently modified.

Fishwick [29],[30] has extended process abstraction concepts and implemented a simulation system that is able to switch between levels within simulation runs. However, although the need for multiple levels of abstraction has been recognized in mainstream AI [39],[92], there has been little consideration of the importance of the morphism concept to this issue. Yet, as tools are needed to enable agents to plan, diagnose, and reason effectively with respect to particular objects, so are tools needed to organize the various models that support such planning, diagnosis, and reasoning. An organized model base enables the agent to deal with the multiplicity of objects and situations in its environment and to link its high level plans with its low level actions. Such a model base is a special case of a multifacetted model base management system [108].

6.4. Levels of Autonomy

The model-based architecture suggests a hierarchy of achievement levels for an autonomous system:

- level 1: ability to achieve prespecified objectives, all knowledge being in the form of models, as in the model-based architecture.
- level 2: ability to adapt to major changes in the environment. This requires knowledge enabling the system to perform structure reconfiguration, i.e., it needs knowledge of structural and behavioral alternatives as can be represented in the system entity structure [109],[114].
- level 3: ability to develop objectives of its own. This requires knowledge to create new models to support new objectives, that is, a modelling methodology [109].

6.5. Conclusion

A multidisciplinary team effort is clearly required to design high autonomy systems. Difficult design issues include trade-offs of function allocation between humans and robots, and between soft (robotic) and hard (conventional control) automation. In the following chapters, systematic design methodologies for control, diagnosis, and task planning are introduced. Their purpose is to attain increasingly higher achievement levels of the high autonomy architecture presented in this chapter.

CHAPTER 7
DEVS-BASED INTELLIGENT CONTROL

7.1. Overview

This chapter describes the development of event-based intelligent control systems by employing the DEVS formalism. DEVS models provide a basis for the design of event-based logic control. In this control paradigm, the controller expects to receive confirming sensor responses to its control commands within predefined time windows determined by its DEVS model of the system under control. In this chapter, the DEVS-based intelligent control paradigm is applied to a space-adapted fluid mixing system capable of supporting e.g. the automation of fluid handling laboratories of the International Space Station Freedom used for research in life sciences, microgravity sciences, and space medicine.

Intelligent control is at the intersection of artificial intelligence, conventional automatic control, and operations research. The intelligent control paradigm is receiving increasing attention in both theory and application [58],[59],[60],[79]. An intelligent control system often employs a hierarchical control structure in which a higher-level intelligent controller supervises a lower-level conventional controller. The event-based control paradigm, introduced by Zeigler [109], realizes such intelligent control by employing a discrete eventistic form of control logic represented by the DEVS formalism [108].

An algorithm to be used by an autonomous system to operate equipment in a partially unknown environment must be robust with respect to modifications of the environment. Unfortunately, the robustness and refinement of a control algorithm are usually in competition with each other. While simple proportional integral (PI) controllers work reasonably well under varying operating conditions, the more refined control algorithms, such as multivariable optimal controllers, are highly sensitive to even small variations in plant parameters[21]. Event-based intelligent controllers are good candidates for implementing more robust co-ordination schemes. The event-based methodology provides a relatively simple and robust real-time control layer that can be linked to both higher level symbolic reasoning layers [89] and lower level traditional control layers.

In conventional sampled-data control, the controller sends commands to the data acquisition subsystem to sample the process at regular intervals. When the sampled value returns, it is stored and tested. Depending on the outcome of the test, a corrective action command is emitted. However, an event-based controller expects to receive confirming sensor responses to its control commands within predefined time windows determined by its DEVS model of the system under control. Since the DEVS formalism is at the heart of event-based control system design, such controllers can be readily checked by computer simulation prior to their implementation. Thus the DEVS formalism plays the same role in event-based control that differential and difference equation formalisms play in conventional control [66]. An advantage of the event-based control logic includes its fault diagnostic capability. Simulation and real-time implementation of DEVS-based control is supported by DEVS-Scheme [68],[96],[104],[108].

The chapter is organized as follows. First it introduces the role of the DEVS formalism in DEVS-based control and formalizes the dynamics of a mixing process as a DEVS model. Then it shows hierarchical modelling of an intelligent control system for space-adapted mixing. This is followed by a discussion of several simulation runs illustrating the technique. Finally, DEVS-based intelligent control is compared with conventional sequential control.

7.2. The Role of DEVS in Event-based Control

For concreteness, we assume that the process to be controlled is a deterministic continuous system modelled using conventional differential equations. Moreover, we assume that the process receives input stimulations that are piecewise constant time functions (sequences of step functions). The approach to be discussed, however, applies to a much wider class of dynamical systems [108]. Indeed, the characteristics of a system which permit it to be represented in the DEVS formalism throw light on the fundamental nature of systems that can be viewed as discrete event systems.

7.3. Example: Space-Adapted Mixing Process

In microgravity all fluid handling must be done without creating liquid-gas interfaces that are not controlled by surface tension. To store fluids, we can use a doubly-contained bottle, in which the inner chamber (an inflatable bag) is kept under constant gas pressure. The basic idea of a space-adapted container is that the inner chamber contains fluid only, whereas the outer container contains the inner container plus compressible gas. When being filled, the inner container expands to adjust to the influx volume, thereby increasing the gas pressure in the outer chamber; conversely, when the inner container is emptied, it contracts, thereby reducing the gas pressure in the outer chamber. Filling and emptying are performed with a syringe that is injected into the chamber through a septum [89].

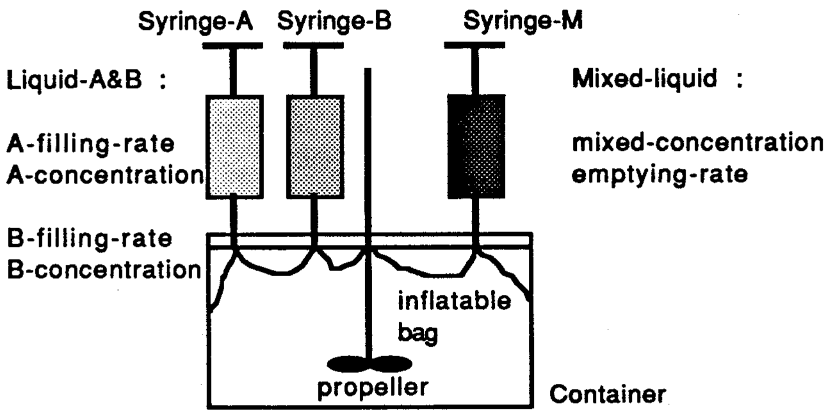


Figure 7.1. Space-Adapted Mixing System

We shall analyze a system consisting of a space-adapted container stirred by a rotating propeller as shown in Figure 7.1. The mixing is accomplished in a batch process - some quantity of fluid with a given concentration is added to a container already filled with a liquid of another concentration. The container is fed with an incoming liquid-A from the syringe-A with a constant flow rate of dissolved material, f_A, by the control command, *fill-A*. When the level of liquid-A reaches its prespecified level, the flow of liquid-A is stopped by the control command, *stop-A*. Then, another liquid-B is added into the liquid-A in the container from the syringe-B with a constant flow rate of dissolved material, f_B, until it receives the control command, *stop-B*. Both feeds contain dissolved materials. The propeller is assumed to start working with a constant mixing rate, α_M, at the same time as liquid-B starts to be filled. The propeller ceases its operation when the control command, *stop-mix*, is issued. When the propeller stops, the concentration of the mix should have reached an equilibrium level. The outgoing flow to the syringe-M has a constant flow rate of mixed material, f_M.

The dissolved material balance equations are as follows:

$$\frac{dV_A(t)}{dt} = F_A(t) - \alpha(t)\, V_A(t)$$

$$\frac{dV_B(t)}{dt} = F_A(t) - \alpha(t)\, V_B(t)$$

$$\frac{dV_M(t)}{dt} = \alpha(t)\ [\, V_A(t) + V_B(t)\,] - F_M(t)$$

where,

$V_A(t)$, $V_B(t)$: volumes of dissolved materials in liquids A and B, which are not mixed at time t, respectively, (cm^3).

$V_M(t)$: volumes of mixed materials at time t, (cm^3).

$F_A(t)$, $F_B(t)$, $F_M(t)$: filling rates of dissolved materials in liquids A and B and emptying rate of their mixed material at time t such that:

$$F_A(t) = \begin{cases} f_A & \text{when liquid-A is filling} \\ 0 & \text{otherwise} \end{cases}$$

$$F_B(t) = \begin{cases} f_B & \text{when liquid-B is filling} \\ 0 & \text{otherwise} \end{cases}$$

$$F_M(t) = \begin{cases} f_M & \text{when mixed liquid is emptying} \\ 0 & \text{otherwise} \end{cases}$$

respectively, (cm^3/sec)

$\alpha(t)$: Mixing rate at time t such that:

$$\alpha(t) = \begin{cases} \alpha_M & \text{when liquids are being mixed} \\ 0 & \text{otherwise} \end{cases}$$

Figure 7.2 illustrates the relationship between dynamic characteristics and corresponding control commands. Figure 7.2(a) and (b) represent input characteristics: filling rates, emptying rate, and mixing rate, respectively. Each input parameter is assumed to have a constant value. Figure 7.2(c) and (d) show output characteristics: volumes of dissolved material and liquid level, respectively. Volumes of dissolved materials exponentially approach their equilibrium values. We adopted the ten percent criterion for reaching an equilibrium level of material concentration, (which can be computed from the concentration of the liquids and their final volumes). Figure 7.2(e) presents corresponding control commands.

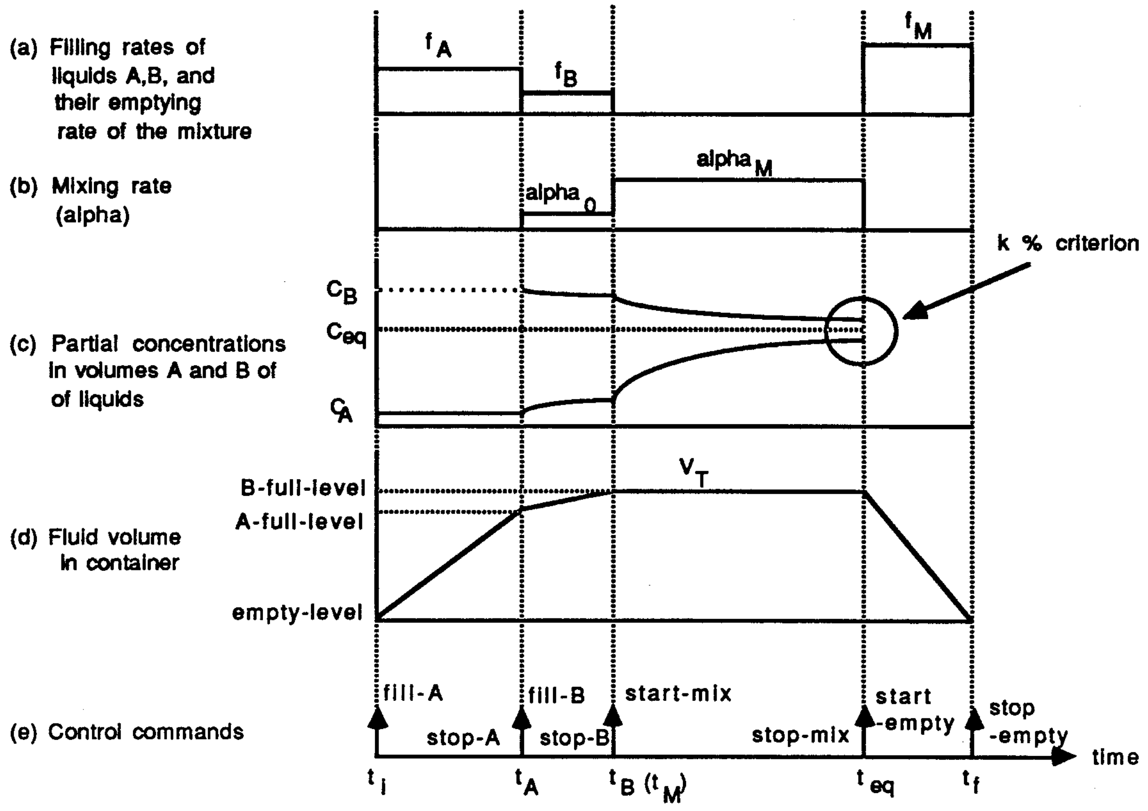
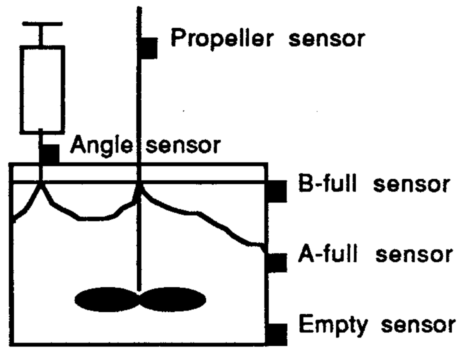


Figure 7.2. Dynamic Characteristics of Mixing System

7.3.1. DEVS Representation of Mixing Process

In DEVS-based control, a DEVS model moves through its check states in concert with external inputs, as long as those inputs arrive within the expected time windows. The idea of employing time windows is well known in computer communication systems, and apparently is also employed in biological systems [85],[89]. Associated with each check state are a minimum lapse time and a time window. In contrast to conventional sampled data systems, event-based logic does not require high sensor output precision. Sensors can have threshold-like characteristics. Only two output states, for example on/off, are needed for each sensor, although more states may be employed. However, to generate the time windows, the output states of the sensors must be accurately and reliably correlated with values of significant process variables.

Figure 7.3 shows possible threshold-type sensors. The empty-sensor, A-full-sensor, and B-full-sensor correspond to specified gas pressure levels in the outer chamber. The propeller-sensor corresponds to a specified speed level. However, some sensors may be difficult, or even impossible, to realize; for example, while the amount of liquids A and B injected into the inner container and the gas pressure of the outer container are easily measurable at all times, some other quantities such as the partial volumes of the liquids A and B and the mixture AB inside the bottle are not easily measurable. Therefore, such quantities must be determined indirectly using state observers. For classical control purposes, state observers are perfectly acceptable and have been in use for years. However, state observers can have a disadvantage when used for fault detection. For example, we can use a model to determine the minimum necessary mixing time as shown in Figure 7.2. We can then use event-based control to schedule both the start-mix and the stop-mix events. However, since we use a state predictor, we don't have a feedback that tells us whether or not the mixing has actually been accomplished. We cannot use the time window approach to detect a faulty state. If, for example, the propellor-sensor measures the rotational speed of the propellor axle outside the bottle, as indicated on Figure 7.3, we may not be able to recognize immediately the situation where the propellor blade has fallen off the axle and lies motionless at the bottom of the container. Therefore, such a fault may not be detected immediately, and will then propagate to a measurable state at a later time. Other quantities can be measured at least indirectly. For example, while we may not be able to measure the volume of the inner container directly, we can easily measure the gas pressure of the outer container, and from there, we can derive the volume of the inner container instantaneously and at all times. For our purposes, we can replace the empty-sensor, the A-full-sensor, and the B-full-sensor by corresponding threshold-type pressure sensors. However, contrary to the classical control problem, in which it is assumed that the controlled process operates flawlessly, in the context of intelligent fault tolerant control, we pay always a price for indirect measurements. For example, a faulty pressure state may indicate a faulty volume state, but it could equally well indicate a gas leak of the outer container. A correct pressure reading may indicate a correct volume state, but it could also be that the bag got perforated, and the fluid and the gas are now mixed.



Propeller sensor	: responds when the propeller speed has reached specified level.
Angle sensor	: responds when angle is between 89 and 91 degrees.
B-full sensor	: responds when pressure has reached specified level.
A-full sensor	: responds when pressure has reached specified level.
Empty sensor	: responds when pressure has reached empty level.

Figure 7.3. Threshold-type Sensors for Space-adapted Mixing System

The DEVS model of the control system performs the control task by changing its state from an initial position on a given threshold sensor boundary to a predetermined and desired sequence, possibly cyclic, of other boundaries. More concretely, this means that we want the system to go through a predetermined sequence of discrete states as reported by sensor readings. The desired sequence of states is determined by the higher level planner. Our control logic will, as each boundary crossing is reached, issue a control action, i.e., send an appropriate input to the system in order to move it toward the next desired boundary. The controller has a time window in which it expects the appropriate sensor states to change to confirm the expected boundary crossing. Time windows are deduced either from previous experience, expert knowledge, previous measurements, abstraction from the so-called DEVS external model [55],[56], or finally simulation of a continuous-time external model [89]. To represent a real (possibly continuous-time) system by a DEVS model in a valid fashion, we must derive the time-advance, internal transition, and external transition functions from the dynamic characteristics of the system.

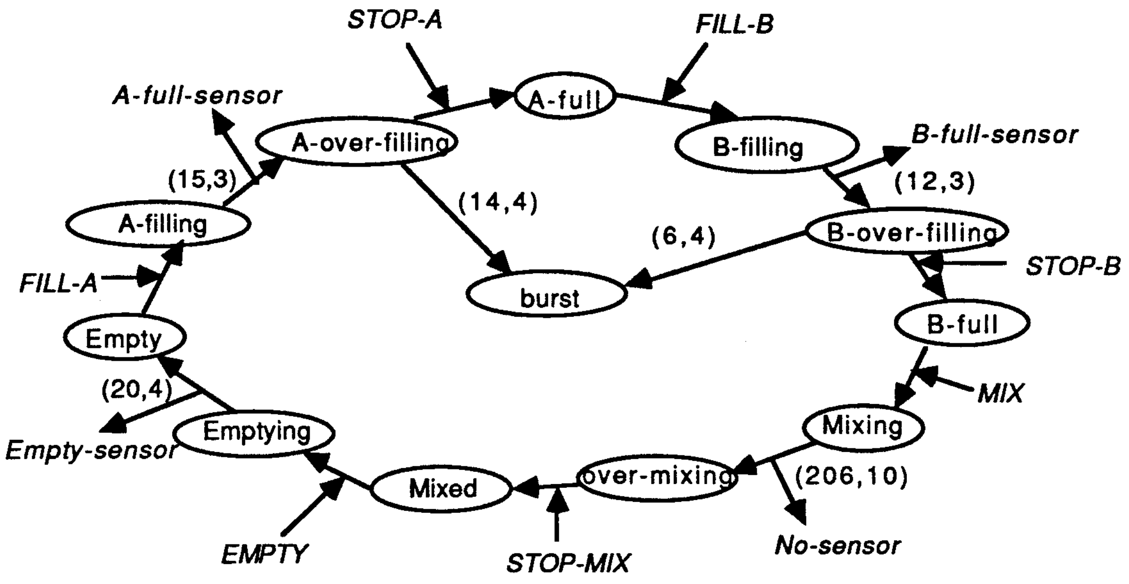
Figure 7.4(a) presents the DEVS states in terms of a partition of the dynamic state space. Given the filling, emptying, and mixing rates we can easily compute the time required to reach to the boundaries of each phase. External transitions occur when the input rates and mixing rate are changed. Internal transitions can occur while the input rates and mixing rate remain constant. Figure 7.4(b) shows the state transition diagram and corresponding control commands. The transition from EMPTY to A-FILLING is caused by an external event, step-increasing the filling rate from 0 to a non-zero value, by control command *fill-A*. The transition from A-FILLING to A-OVER-FILLING is an internal event whose time advance corresponds to the A-filling-time. This can be directly obtained from a series of continuous system simulation runs by varying the disturbances and system parameters of a process model in accordance with normal operating conditions [89]. The transition from A-OVER-FILLING to A-FULL is an external event caused by shutting off the influx, by control command *stop-A*. Mutatis mutandis for the other transitions.

Figure 7.4(b) represents only one of many possible plans of operation (sequences of control commands), namely, a simple plan in which all actions occur sequentially without overlap, although it is possible to overlap actions in many ways. One plausible alternative is to let filling of the second liquid and mixing proceed concurrently. The expanded DEVS model that includes this alternative is shown in Figure 7.4(c). We would expect this alternative to save time due to earlier mixing. Later it will be shown how a planner can select an optimal path from an initial state to a goal state.

Empty	:	level <= empty level
A-filling	:	level <= A-full level, A-filling-rate > 0
A-over-filling	:	level > A-full level, A-filling-rate > 0
Burst	:	liquid overflow, level > TOP, A-filling-rate > 0 or B-filling-rate > 0
A-full	:	level > A-full level, A-filling-rate = 0
B-filling	:	A-full level < level <= B-full-level, B-filling-rate > 0
B-over-filling	:	level >= B-full level, B-filling-rate > 0
B-full	:	level > B-full level, B-filling-rate = 0
Mixing	:	Concentration > 10% criterion, mixing-rate > 0
Over-mixing	:	Concentration <= 10% criterion. mixing-rate > 0
Mixed	:	Concentration <= 10% criterion. mixing-rate <= 0
Emptying	:	empty level < level < B-full level, emptying rate > 0

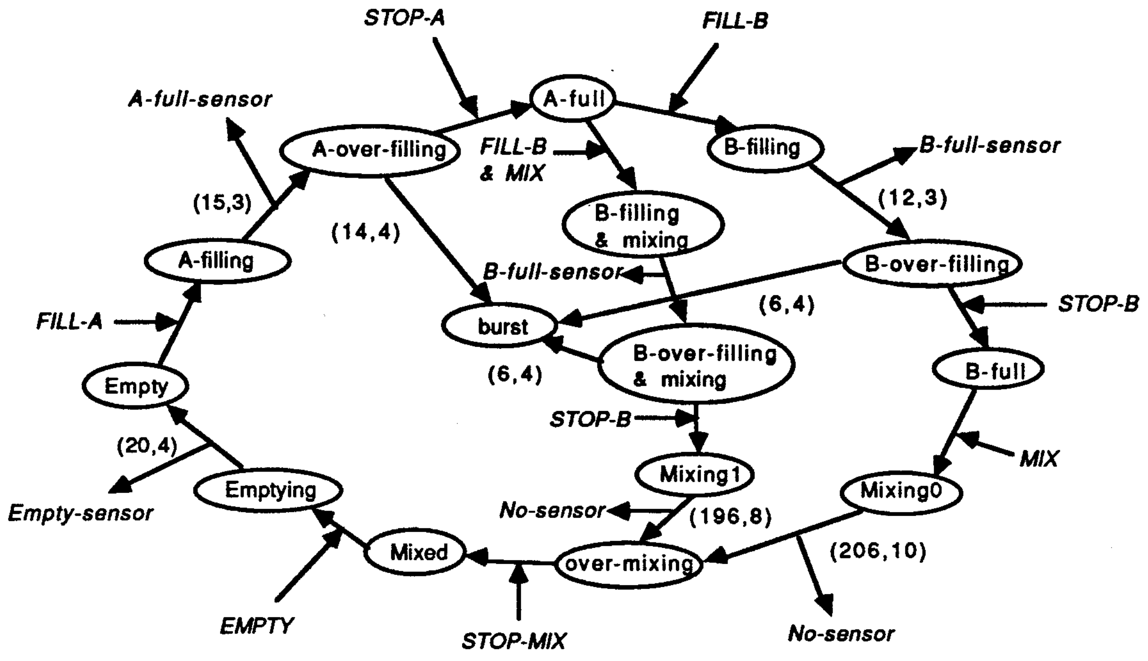
where, DEVS model state is (q,x) such that q(state) = {level, concentration}
and x(input) = {A-filling-rate, B-filling-rate, mixing-rate, emptying-rate}

(a) DEVS State Definitions



Where () = [Tmin, Twindow]

(b) State Transition Diagram



(c) Possible Alternative of State Transition Diagram

Figure 7.4. DEVS State Definitions and State Transition Diagrams

7.3.2. Modelling and Simulation Approach

The model specification and retrieval in the DEVS-Scheme simulation environment are mediated by a knowledge representation component designed using the system entity structure concept discussed in Chapter 2 [95],[104]. Figure 7.5 shows a system entity structure for the event-based mixing process control.

To achieve realism, models of intelligent control units must be able to represent not only their decision making capabilities but also the models of the real system, that the decision making component uses to arrive at its decision.

The simulation test is decomposed into a model representing the real bottle, MIX-E, the so-called external model of the system, and a coupled model representing the control-unit. The control-unit is decomposed into an operator for filling, mixing, or emptying a container and a diagnoser for discovering the causes of any operational faults.

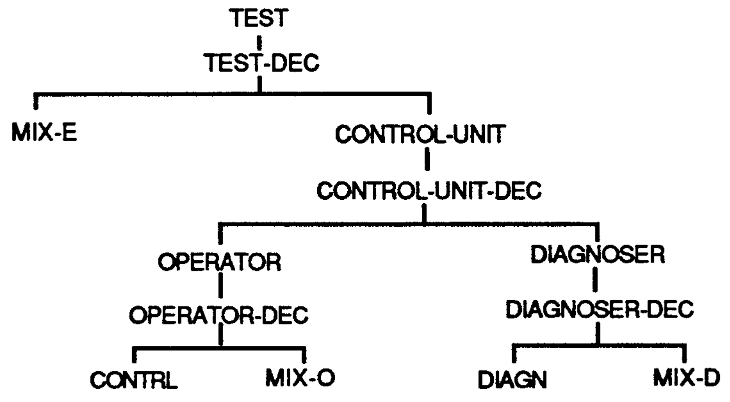


Figure 7.5. System Entity Structure for a Mixing System

There are three types of mixing models used in the simulation that are related to each other by abstraction:

- MIX-E: external model of the space-adapted mixing system. MIX-E is able to respond to both operational commands and diagnostic probes. It is external to the controller.
- MIX-O: operational model of the mixing system used by a controller, CONTRL, to generate its commands and verify the received sensor responses.
- MIX-D: diagnostic model of the mixing system used by a diagnoser, DIAGN, to determine the probable source of a breakdown. It is a classification, or expert-system-like model.

In this configuration, the external model, MIX-E, is able to respond to both operational commands and diagnostic probes. The advantage in folding different submodels into one as we have done, is the straightforward sharing of common parameters and state variables. In contrast, distinct models incur communication overhead required to synchronize such shared variables.

The various models of the same system can all be derived from the same base model, which represents the continuous system, by abstraction processes [56],[89],[102]. Figure 7.6 illustrates an abstraction sequence for models of the mixing process. The base model, MIX describes a continuous process as discussed in the previous section. From the base model, we shall construct a DEVS model, MIX-E, to serve as the external model of the mixing process in a discrete event simulation. We shall also derive a DEVS model, MIX-O, from MIX-E, which serves as the internal model used to operate the space-adapted container. These models are related by abstraction, i.e., by a form of homomorphic relation where MIX-O is an abstraction of MIX-E, which in turn is an abstraction of MIX.

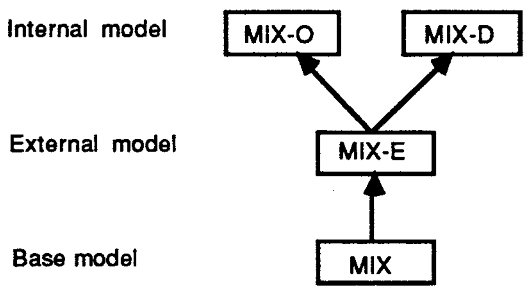


Figure 7.6. Abstraction Sequence for Models of Mixing Process

Figure 7.7 shows a layered architecture of the mixing control system. Level III is the lowest level, where the real system under control exists. The external model, MIX-E, represents the real system in our simulation. It receives input messages such as control commands and read-sensor commands from the next higher level, Level II. It sends the sensor readings to the next higher level. Level II corresponds to the control unit shown in Figure 7.5. It has an abstract model, MIX-O, and a rule-based model, MIX-D, to provide the necessary pieces of information for the controller and diagnoser, respectively. It also sends a result message to its next high level, Level I. We call the level "goal agent". It corresponds to the highest unit of the control system. Notice that the three level hierarchy shown on Figure 7.7 is not the same as the three-layer hierarchy discussed in Chapter 6. It will be shown later how these different types of hierarchy are mapped onto each other.

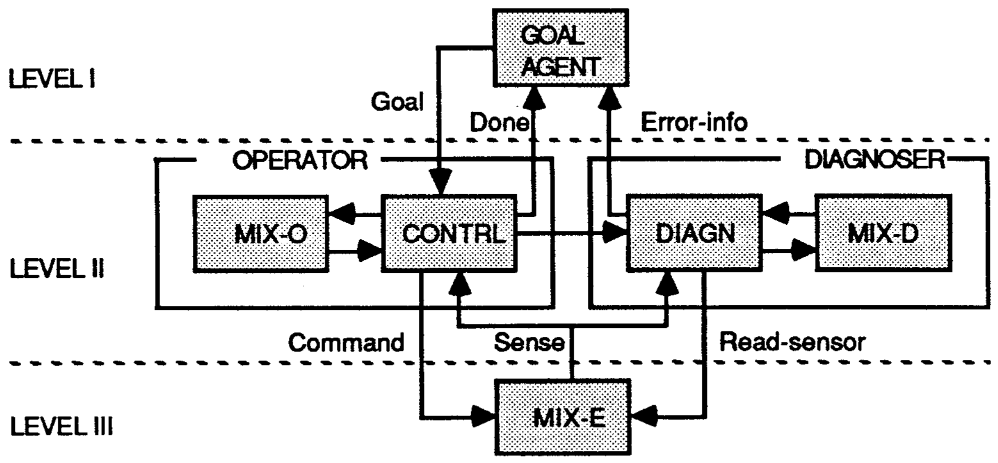


Figure 7.7. Layered Architecture of the Mixing Control System

External Model : MIX-E

The external model, MIX-E, can be built to represent a few of the likely problems that might arise in operation. For example, it allows the syringe to be inserted askew, the tube be blocked or leakage to occur through the inner chamber wall. These anomalous conditions can be imposed on the model by assigning it appropriate parameter values. The controller that operates the container will detect abnormal behavior during operation and diagnose its source as one of these anomalies.

To implement MIX-E, we use the DEVS-Scheme class *forward-models* to conveniently express the transitions and outputs in the form of sensor responses. *Forward-models*, a specialized class of *atomic-models*, generates model objects that inherit all the instance variables and methods of *atomic-models* (see Figure 3.1). In addition, *forward-models* contains a forward-chaining inference engine that facilitates writing models in a rule based manner. In general, internal transition rule conditions test the execution phase and the state variables of the model while external transition rule conditions test also the input content structure and elapsed time. Once its condition is true, the triggered rule becomes a candidate to execute its action and then to change the state of the model [109].

Internal Model : MIX-O

Having derived an external DEVS model of a continuous system, we are in a position to further abstract it for use as an internal model within an intelligent agent. For this purpose we introduce another means of specifying *atomic-models*, the sub-class *table-models*. The information required to specify the transition and other functions of an *atomic-model* is provided in the form of tuples in a relation, called the transition table. For example, as shown in Figure 7.8(a), the first assertion of such a tuple states that if the current state is EMPTY and the input is *fill-A* then the next-state is A-FILLING; also the output in the current state is nil and its time-advance is infinity (ta = inf). The window is given by two fields: next-ta(21) gives its lower bound, and next-wind(4) gives its width (so its maximum bound is 25). Every *table-model* has (beside from the inherited variables, phase and sigma) state variables for the transition table, a goal table, and a time window. As with *forward-models*, the internal and external transition functions, and output function of *table-models* implement an interpreter of the transition table.

Let us consider the approach taken to construct the external and internal operational models, such as MIX-E and MIX-O, respectively. Having the discrete event model, MIX-E, we can derive a *table-model*, MIX-O suitable for use by a generic controller to be discussed later. MIX-O is intended to be a simplified version of MIX-E that is valid in the region of normal operation of the system. More specifically, MIX-E and MIX-O are related by an approximate homomorphism, in which corresponding states have the same outputs and transition to corresponding states; only the time advance values of corresponding states may differ - however this divergence must lie within predefined time windows. The time window associated with a state in a *table-model*, MIX-O, is determined by bracketing the time advance values of all internal transitions associated with the corresponding states in its parent model, MIX-E. Thus divergence arises due to variations in parameters and initial states considered to represent normal operation [14],[109].

A *table-model* must have a method, *input?*, which returns the next input required to reach a goal from the current state. Another method, *plan*, is required to figure out what this input should be. The planner produces a time optimal path from an initial state to a goal state. Since discrete event models embody timing it is natural to base optimal sequencing on predicted execution times. The result is stored as a relation, the goal-table, as shown in Figure 7.8(b).

The planner works by developing paths backward from the goal until the given initial state (the starting state of the controlled system) is reached. As each state is encountered, an entry is made in the goal table for it with the time-to-reach-goal entered. If a state is encountered again, with a smaller time-to-reach-goal, a new entry replaces the existing one; otherwise no change is made. When such a replacement occurs, any predecessors of the state in the goal table must have their time-to-reach-goal downward adjusted accordingly. This is a form of dynamic programming [65] based on the principle that every subpath of an optimal path must itself be optimal. Thus we need to keep track only of the input required to continue an optimal path from any state, not the whole path itself.

In Figure 7.8(b), we see that the planner has assigned the command *fill-B & mix* to be given from state A-FULL as the quickest way to reach the MIXED goal. This reflects the fact that concurrent mixing and filling is faster than sequential operation as derived from the underlying continuous model.

state	input	next-state	output	ta	next-ta	next-wind
empty	fill-A	A-filling	()	inf	15	3
A-filling	()	A-over-filling	A-full-sensor	15	14	4
A-over-filling	()	burst	()	14	inf	()
A-over-filling	stop-A	A-full	()	14	inf	()
A-full	fill-B	B-filling	()	inf	12	3
A-full	fill-B&mix	B-filling&mixing	()	inf	12	3
B-filling	()	B-over-filling	B-full-sensor	12	6	4
B-filling&mixing	()	B-over-filling&mixing	B-full-sensor	12	6	4
B-over-filling	stop-B	B-full	()	6	inf	()
B-over-filling	()	burst	()	6	inf	()
B-over-filling&mixing	stop-B	mixing1	()	6	196	8
B-over-filling&mixing	()	burst	()	6	inf	()
B-full	mix	mixing0	()	inf	206	10
mixing0	()	over-mixing	no-sensor	206	10000	20
mixing1	()	over-mixing	no-sensor	196	10000	20
over-mixing	stop-mix	mixed	()	10000	inf	()
mixed	empty	emptying	()	inf	20	4
emptying	()	empty	empty-sensor	20	inf	()

(a) State Table

state	goal	input	time-to-reach-goal
empty	mixed	fill-A	223
A-over-filling	mixed	stop-A	208
A-full	mixed	fill-B&mix	208
B-over-filling&mixing	mixed	stop-B	196
B-full	mixed	mix	206
over-mixing	mixed	stop-mix	0
mixed	empty	empty	20

(b) Goal Table

Figure 7.8. Table Specifications of an Internal Model, MIX-O

Diagnostic Model : MIX-D

The diagnostic model, MIX-D, employed by the diagnoser inference engine, is derived from the external model, MIX-E, but not in a homomorphic fashion. Indeed, the diagnostic model is an inversion process going from external effects to underlying causes. The validity criterion for the diagnostic model is thus the following: the occurrence of every fault representable in the external model, MIX-E, should be identifiable by the diagnostic model, MIX-D, under the operation of the diagnosing inference engine. The diagnostic model, MIX-D, is implemented in the *forward-models* class in DEVS-Scheme. Some of the diagnostic rules for MIX-D are given in Figure 7.9.

If A-full-backup-sensor is not true and A-full-sensor is true,
then "A-full-sensor is bad".

If B-full-backup-sensor is not true and B-full-sensor is true,
then "B-full-sensor is bad".

If angle-sensor indicates off-angle or tube-angle of vision
is less than 89 or greater than 91,
then "Tube is off-angle".

If timing is too-late and phase is A-filling and level is
less than 100,
then "Tube is off-angle or leaky".

If tube-constriction of vision is greater than 50,
then "Tube is constricted".

If empty-sensor is true and level is greater than 1,
then "Empty-sensor is bad".

If indicator-sensor is A-full-sensor and indicator-timing is too-late
and phase is A-filling and level of vision is less than 100,
then "Tube is leaky".

Figure 7.9. Fault Diagnosis Rules for MIX-D (partially-shown)

Controller Model : CONTRL

The controller is a generic engine similar to an inference engine that can operate on any suitable DEVS model of type *table-model*, such as MIX-O. Using its goal table, the controller can obtain information from the model relating to commands. Using its state table, it can obtain information from the model relating to expected response times and their time windows. The controller issues these commands to the controlled device. When proper response signals are received, the controller causes the model to advance to its next state corresponding to the one the device is supposedly in. Thus if the model is valid and if the operation proceeds normally, the underlying homomorphic relation between the model and the controlled device is maintained. The controller ceases interacting with the device as soon as any discrepancy occurs in this relationship and calls on a diagnoser to figure out what has happened. The controller model, CONTRL, is implemented in the *forward-models* class of DEVS-Scheme.

Diagnoser Model : DIAGN

The operator and diagnoser are coupled so that once the controller has detected a sensor response discrepancy, the diagnoser is activated. Data associated with the discrepancy, such as the phase in which it occurred, and its timing (TOO-EARLY, TOO-LATE), are also passed on to the diagnoser. From such data, as well as the information it can gather from auxiliary sensors, the diagnoser tries to discriminate the fault that occurred. Figure 7.10 shows various data types for fault diagnosis. The indicator used by the controller keeps track of the container states estimated by the level-sensors. However, auxiliary sensors such as a backup pressure (volume) sensor, an angle sensor, and a vision sensor provide more accurate container state information. The diagnosis can be forwarded to a unit that tries to restore the system to normal operation. For example, if the verification from the A-FULL sensor was TOO-LATE during A-FILLING, the diagnoser might determine that the syringe was most likely misaligned and the repair unit would then try to correct the alignment. We shall not discuss modelling of the repair process here.

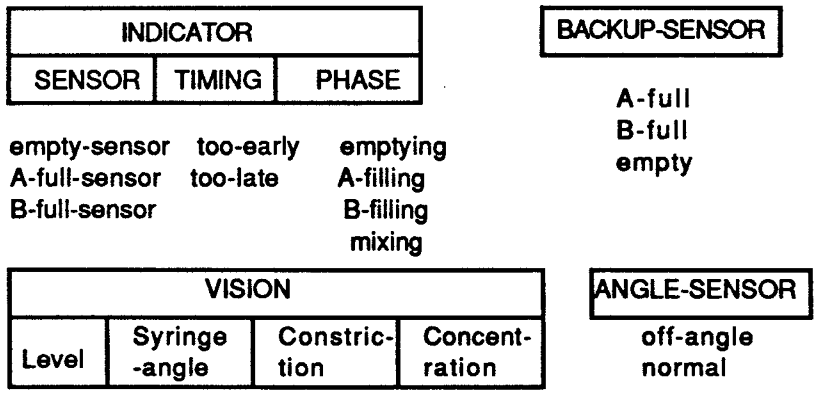


Figure 7.10. Various Data Types for Fault Diagnosis

7.3.3. Simulation Results

In the DEVS-Scheme environment, a system entity structure such as the one shown on Figure 7.5 can be transformed into a simulatable test mockup. In this mockup, we can set the external model, MIX-E, into an initial state and choose a parameter setting that represents either a normal or an abnormal condition. We set the internal operational model, MIX-O, to a corresponding phase, assign a goal phase to the control model, CONTRL, and start the simulation. If MIX-E is started in a normal condition, the simulation should run to the point where CONTRL achieves its goal. If started in an abnormal condition, CONTRL should detect this at some point due to a time window violation and activate DIAGN, which should eventually conclude the responsible fault in agreement with the one introduced at the start of the simulation. Figure 7.11 shows initial values for an external model, MIX-E, under normal operation. We assume several time delays of sensor readings, for example 1 sec for the backup-sensor reading and 100 sec for the visual-sensor reading. These time delays can be taken into account when deciding which sensors to interrogate [80].

A-filling-rate	7	: Filling rate of liquid-A
B-filling-rate	5	: Filling rate of liquid-B
emptying-rate	8	: Emptying rate of mixed liquid
A-full-level	100	: Expected level when liquid-A is full
B-full-level	167	: Expected level when liquid-B is full
empty-level	1	: Initial level of the container
angle	90	: Syringe angle
constriction	5	: Constriction effect
leakage-rate	0.001	: Leakage rate of the container
A-concentration	0	: Initial concentration of liquid-A
B-concentration	50	: Initial concentration of liquid-B
alpha	1	: Mixing effect coefficient

Figure 7.11. Initial Values for the Simulation Test

The simulation results under normal operation for each goal plan are shown in Figure 7.12(a), (b), and (c), respectively. Figure 7.12(a') illustrates the simulation results of a faulty case for a goal plan to proceed from state EMPTY to state A-FULL.

Under normal operation, as shown in Figure 7.12(a), CONTRL issues the control command, *fill-A*, to the external model, MIX-E, and also to the internal model, MIX-O, and then waits for the sensory response to arrive within the scheduled time window of 4 sec. If the sensory response arrives within the prescribed time window, the controller generates the next command, and so on, until MIX-E reaches its goal state. However, in the faulty case, where the tube is leaky, the leakage rate is set to 1 instead of 0.001. As illustrated in Figure 7.12(a'), no response is received from MIX-E prior to time step 25, the maximum time. Therefore, the controller generates the error command, *error*, to the diagnoser. The diagnoser, DIAGN, checks the data associated with the discrepancy, such as the phase in which it occurred and its timing by referring to MIX-E. The expert-system-like model, MIX-D, concludes that "the tube is leaky" by using the data from DIAGN. Because the indicator shows that its sensor is A-FULL-SENSOR, its timing is TOO-LATE, and its current phase is A-FILLING, but the visual sensor shows that the liquid level has not reached its specified level, 100 (see Figure 7.9).

Clock-	MIX-E		MIX-O		CONTRL	
time	state	output	state	output	state	output
0	(A-filling 16)	()	(A-filling)	(fill-A 15 3)	()	(Fill-A)
15	()	()	()	()	(window 3)	()
16	(A-over-filling 16)	(A-full-sensor #T)	(A-full)	(Stop-A)	(Check 1)	()
17	(A-full)	()	()	()	()	(Stop-A)

(a) Goal Plan : EMPTY -> A-FULL

Clock-	MIX-E		MIX-O		CONTRL	
time	state	output	state	output	state	output
17	(B-filling 12) & mixing	()	(B-filling & mixing)	(fill-B 12 3) &mix	()	(Fill-B &mix)
29	()	()	()	()	(window 3)	()
30	(B-over-filling 7) &mixing	(B-full-sensor #T)	(Mixing1)	()	(Check 1)	()
30	(Mixing1 201)	()	(Mixing1)	(Stop-B 196 8)	()	(Stop-B)
226	()	()	()	()	(window 8)	()
235	(over-mixing 10000)	()	(Mixed)	(Stop-mix)	(Check 1)	()
236	(Mixed)	()	()	()	()	(Stop-mix)

(b) Goal Plan : A-FULL -> MIXED

Clock-	MIX-E		MIX-O		CONTRL	
time	state	output	state	output	state	output
236	(Emptying 21)	()	(Emptying)	(Empty 20 4)	()	(Empty)
256	()	()	()	()	(window 4)	()
256	(Empty)	(Empty-sensor #T)	(Empty)	()	(Check 0)	(Finished)

(c) Goal Plan : MIXED -> EMPTY

Clock-time	MIX-E		MIX-O		CONTRL		MIX-D		DIAGN	
	state	output	state	output	state	output	state	output	state	output
0	A-filling 19		A-filling	Fill-A 15 3		Fill-A				
15					Wind 3					
18					Error	Too-late			Start 1	
19	Read-ing 1								Wait-sensor	
20	Read-ing 10	In-between							Wait-sensor	
30	Visual 100	Normal							Wait-sensor	
130	Finish -read 1	Vision -info.							Wait-sensor	
131	Pass-ive	Done							Start-diagn 1	Sensor -data
132							Pass-ive	Tube is leaky	Pass-ive	Tube is leaky

(a') Goal Plan : EMPTY -> A-FULL (fault case)

Figure 7.12. Simulation Results

7.4. Discussion

The DEVS formalism was introduced in the 1970's to help specify simulation models in a rigorous, system-theory based manner [108],[115]. Since then, other approaches to modelling DEDS (Discrete Event Dynamic Systems) have emerged. Most are algebraic or graphical in character and do not include the time element that DEVS inherits from its system theoretic origins. The most closely related formalisms are found under the framework of Generalized Semi-Markov Process (GSMP), in which we can include the stochastic generalizations of Petri Nets [77]. GSMP, as formulated by Glynn [35] and

Cassandras and Strickland [9], attempt to formalize discrete event simulation models as Markov processes with countable state sets that are amenable to mathematical analysis. However, Zeigler [109] showed that fundamental DEVS behaviors require a continuous state space and cannot be expressed by tractable GSMPs. Indeed, merely having a component keep its scheduled event time despite interruption is impossible without a continuous state set. Zeigler [109] concludes that DEVS appears to be more powerful than GSMP, trading mathematical tractability for expressive power.

At the process control level, limitations have been recognized in recent years in the ladder specification technique used by programmable logic controllers (PLC), an approach that is wide spread in industry. Ladder networks are limited to combinatorial logic, are difficult to understand, and provide an inadequate basis for fault recovery [54]. In the mid 1980s, artificial intelligence and expert systems were being suggested as approaches to overcome such limitations [58],[59],[60],[79],[87] stimulating the knowledge-based approach adopted in our DEVS-based intelligent control methodology [102]. Independently of our efforts, methods of low level programmable control specification were also being developed. In particular, the GRAFCET language [47],[54] provides a graphical specification of sequential dynamic behavior similar to Petri nets and other conventional state-transition diagrams [20],[38],[116]. The strengths and limitations in expressive power of GRAFCET are discussed in [26] with the motivation to develop more expressive graphical languages for PLC-based process control and real-time parallel computing applications.

In contrast to such specification languages, the DEVS methodology does not aim to provide improved graphical sequential logic specification languages. First of all, it resides at a more abstract level as a mathematical, set theory based formalism. Moreover, as indicated, the DEVS formalism has been shown to have full expressive power for discrete-event simulation modelling [108],[109],[115], and it is capable of providing sound semantics for such graphical languages. Recent work has also shown how the event-based control systems developed in DEVS-Scheme can be readily migrated to actual real-time operation by interfacing the simulation environment with the plant and driving the model state transitions with a real-time clock [96].

The emphasis in this chapter is on application to intelligent control where low level conventional controllers are being interfaced with high level symbolic decision making [102]. Such "hybrid" modelling methodology has been shown to be an important problem in the theory of autonomous and intelligent control [6]. Indeed, to design intelligent, autonomous systems, many control aspects come into play, such as planning, operation, diagnosis, and recovery. Graphical sequential specification languages are concerned mainly with normal operation. While an improved language such as GRAFCET facilitates maintenance and trouble shooting, the DEVS approach goes one step further: it is concerned with an integrated approach in which models can be developed to support all control aspects.

Generally, a state correspondence between base and lumped models must be preserved under transitions and outputs. Morphisms differ in such details as the nature of the state correspondence and the lengths of microstate transition sequences corresponding to macrostate transitions [108],[115]. Choice of a particular morphism establishes the criterion of validity for simplification. In our approach to model development based on such morphisms, a relatively complex simulation base model is abstracted into simplified models that are oriented to the needs of a particular task such as planning, operation, or diagnosis. Strategies derived from simplified models can be tested against the base model by simulation before actual implementation.

This approach needs some discussion as objections have been raised that it consists of degrading the mathematical information one has on the system and is completely at odds with what the standard control theorists are wont to do: one of their constant goals being to improve the mathematical description of the control problem. In answer, we emphasize that base model formulation is entirely in keeping with the attempt to improve mathematical description: here one wants to be as rigorous and comprehensive as necessary to deal with all of the control aspects of interest. However, for application to a particular aspect, such as operation, it is advantageous to abstract the base model to a simplified lumped model that provides crucial computational complexity reduction especially as needed under real-time operational constraints. Of course, the model should not be arbitrarily "degraded": indeed, the criterion for assuring that a lumped model remains valid and exhaustive within the scope of its intended operation is the homomorphism criterion just mentioned.

Although much research remains to be done on the abstraction of continuous base models to discrete-event lumped models, some preliminary answers can be given to more fundamental questions:

How does the size of the discrete-event "automaton" increase with the number of sensors and actuator thresholds? As indicated, the partitioning of the continuous state-space is determined by the output map, which is contained in the cross-product of the sensor states. In theory, all such combinations of sensor states are possible, but in practice, due to correlations and redundancy, a relatively small percentage will occur in normal operation. Sensor states should therefore be introduced only as needed so that, using a minimal sensor set, the resulting partition represents the coarsest valid continuous state reduction.

Will such a combinatorial dependency not lead to very large automata in realistically dimensioned examples? This would be true if a monolithic approach were adopted. In contrast, as indicated earlier, the DEVS formalism supports the specification of coupled models, closure under coupling, and the resulting hierarchical, modular model construction capability. Thus, complex continuous systems can be hierarchically decomposed to a level where the resulting atomic components can be represented as discrete-event systems in the manner just described. The resulting control structure is a hierarchical one in which higher level supervisors coordinate the actions of lowest level controllers. Concurrency in operation is naturally handled in such a representation as opposed to the graphical state-based approaches mentioned above. Description of such hierarchical control is described in Chapter 10.

Can the generation of the DEVS automaton from the continuous representation be automated? As described earlier, several approaches can be taken to such generation. The DEVS-Scheme environment has been extended to support continuous model specification and the automatic abstraction of event-based control models including the essential time-window information [55],[56],[89]. Of course, the user must specify the sensor partitions and the parameter variations characterizing normal operation, but from there on the DEVS model is algorithmically generated.

If one is interested in diagnostics, does abnormal behavior introduce new states? We have indicated that event-based control provides TOO-EARLY/TOO-LATE information that can be used in the diagnostic process. However, this is a side-benefit that need not result in increased states of the operational model. This is so since the lumped operational model represents normal behavior and does not in itself include diagnostic aspects. It is the base model that must be sufficiently refined to represent all abnormal behavior of interest. The base model abnormal behavior information is mapped into the diagnostic model, dimensioning its state space, so to speak, rather than that of the normal operation model.

How does the complexity of simulation and diagnostic ability compare with standard techniques based on estimator, observer and decision theory ? The estimator concept, available to standard linear control theory, is needed where the process state must be identified in the presence of noise. In this paper, we are assuming noise free, highly non-linear systems so that the comparison is not germane. However, general system reduction and morphism concepts underlie both approaches [51],[57],[93].

7.5. Conclusions

This chapter has shown how suitably operating on the structure of such DEVS models provides a basis for design of event-based logic control. In this control paradigm, the controller expects to receive confirming sensor responses to its control commands within predefined time windows that are determined by the DEVS model of the system under control.

A space-adapted mixing control process was advantageously represented as a family of discrete event models by employing techniques based on the DEVS formalism. Several fluid handling models have been successfully tested in the DEVS-Scheme environment.

An essential advantage of DEVS-based control is that the error messages it issues can bear important information for diagnostic purposes. This possibility arises when a DEVS model is developed for the process and used to determine the time windows for sensor feedback. As a side benefit, causes for other responses may also be deduced.

Since the DEVS formalism is at the heart of event-based control system design, such controllers can be readily checked by computer simulation prior to implementation. Thus the DEVS formalism plays the same role with respect to event-based control that differential and difference equation formalisms play to conventional control. This principle and its applicability implemented using the DEVS-based control paradigm were illustrated by means of a space-adapted fluid mixing system capable of supporting the automation of fluid handling laboratories as they will be built as part of the International Space Station Freedom to be used for research in life sciences, microgravity sciences, and space medicine.

CHAPTER 8
MODEL-BASED TASK PLANNING

8.1. Overview

Originally, task planning developed as a subfield of artificial intelligence. However, task planning has yet to find a coherent theory. Moreover, task planning should not be considered independent of task execution. Recently, task planning has been referred to in the context of a variety of techniques ranging from hierarchical control of robots to the solution of constraint satisfaction problems to expert systems.

The automation of reasoning about actions is useful in many practical problems with significant commercial potential. Research on reasoning about actions is generally published and presented as part of a subfield of AI labeled "planning." Nilsson originally specified problem solving and planning as being one of the four fundamental application areas of AI. However, the *weak methods*, employing little domain knowledge, originally used in AI for problem solving and planning, proved inadequate for tackling complex real world problems. Thus in seeking solutions in this area, larger amounts of knowledge have since been utilized. The net result has been that the knowledge engineering methodology used for expert systems has been adapted for use in problem solving and planning. Thus the boundary between problem solving, planning, and expert systems has faded [34].

Most AI applications can be considered as examples of problem solving, which are well covered in other AI application areas such as expert systems, computer vision, and language understanding. Planning can be defined as the design process for selecting and stringing together individual actions into sequences in order to achieve desired goals [34]. It also can be viewed as reasoning about how actions affect the world [92]. From the point of view of learning systems, planning can furthermore be defined as the integration of a model of dynamic memory, learning, and planning into a single system that learns about planning by creating new plans and analyzing how they interact with the world [37],[73].

Faced with the overwhelming complexity of planning, a model-based approach strives to integrate key ideas of existing planning paradigms; the hierarchical abstraction approach (SIPE) [92], the time constraints approach (DEVISER) [88], and the dynamic memory retrieval approach (CHEF) [37],[73].

Model-based planning can provide an extended planning paradigm, in which it can support not only the initial planning problem (searching problem) but also the dynamic memory retrieval problem (representation problem). Moreover, it provides dynamical reaction with a simulation (execution) through planning, operation, diagnosis, and repair [13].

This chapter introduces an extended planning paradigm, in which existing paradigms can be integrated with some focus on a model-based approach. This model-based planning methodology is described in detail to develop high autonomy system. The chapter is organized as follows. First it shows a model-based planning methodology using the SES/MB environment. It then introduces an abstraction process to build a hierarchical planning/execution structure coherently as an extended planning paradigm.

8.2. Viewing Planning as a Pruning Activity

In our model-based planning employing the SES/MB environment, the pruning of an SES to select a PES from alternatives can be viewed as an initial planning process. This pruned entity structure (initial plan) is in turn transformed into a simulation model (for execution). The PES may be catalogued to facilitate subsequent recognition and retrieval (experienced plan). The SES/MB environment with the DEVS formalism supports various concepts useful in the context of planning such as causality, time constraints, and nonlinearity [13],[76],[88],[92]. Figure 8.1 illustrates an analogy between existing planning issues and the SES/MB environment. A more detailed discussion is available in [13],[17].

Key issues of existing planning systems	*SES/MB environment*
Hierarchical	Hierarchical, modular system design
Time constraint	Handled by event-based control logic within SES/MB environment
Causal theory	MB simulator includes a time scheduler such as time-of-next-event
Nonlinear (parallel planning)	Layered intelligent agent can handle nonlinear and/or parallel planning
Replanning during execution	Repruning of SES during simulation (variable structure)
Dynamic memory (save and retrieve): indexing of resultant plan	Save/retrieve to/from ENBASE: cataloguing of PES
Domain-independent	SES is based on its formalism
Extended paradigm: self-diagnostic, self-repairing	integrated in event-based control unit

Figure 8.1. Analogy Between Existing Planning Systems and SES/MB Environment

8.3. Model-based Planning Approach

An autonomous, intelligent system needs not just one, but many, models on which to base its operation, diagnosis, repair, and planning activities [6],[58],[105]. These models differ in levels of abstraction and in formalism. Concepts and tools are needed to organize the models into a coherent whole. An organized model base helps the intelligent formalism. Concepts and tools are needed to organize the models into a coherent whole. An organized model base helps the intelligent unit to cope with the multiplicity of objects and situations in its environment and to link its high level plans with its actual low level actions.

In the model-based approach, the planner first retrieves from the memory, ENBASE, a plan that might be used to achieve a given goal. It then generates a new trial plan if no existing plan is suitable. In the sequel, this candidate plan is projected forward via simulation to observe the outcome. This outcome is finally examined to see if there are any conflicts with other goals if this plan is pursued.

In planning a course of action to achieve a set of goals, time is often a key parameter. The need to account for time explicitly in AI planners has been recognized for a long time. Vere's DEVISER synthesizes plans to achieve goals that may have time restrictions on when sets of goals should be achieved and on how long the goal conditions should be preserved [vere]. The planning paradigm with time constraints can also be accomplished during execution (simulation) within the model-based approach. The time window and duration (as discussed in the event-base control of Chapter 7) can be directly obtained from a series of continuous system simulation runs by varying the disturbances and system parameters of a process model in accordance with normal operating conditions [wang-cell]. In the planning hierarchy, these segmented time windows should be aggregated according to the abstraction level.

8.4. Model-based Planning Methodology

Once the DEVS model base is developed and a system entity structure has been constructed on the basis of the model-based hierarchical system structure, we should know how to relate such a structure with a given goal. In other words, once we know locations (structural states) and characteristics (behavioral states) of resources (models), we have to know how to allocate (plan) resources to accomplish a task (goal). Figure 8.2 denotes a planning concept to build autonomous systems.

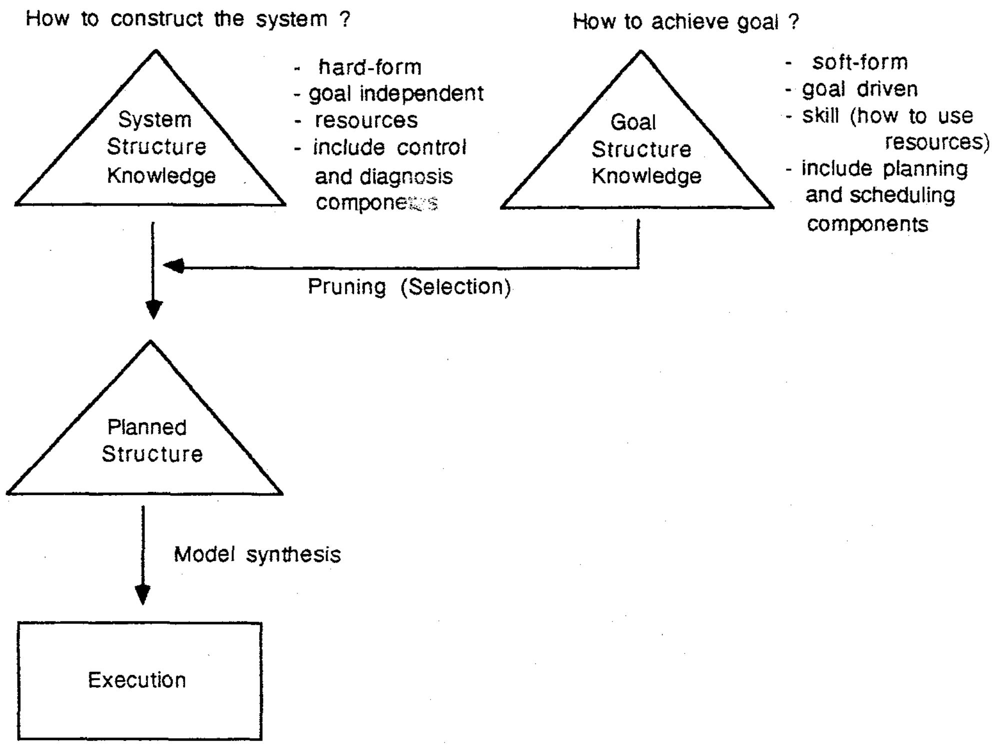
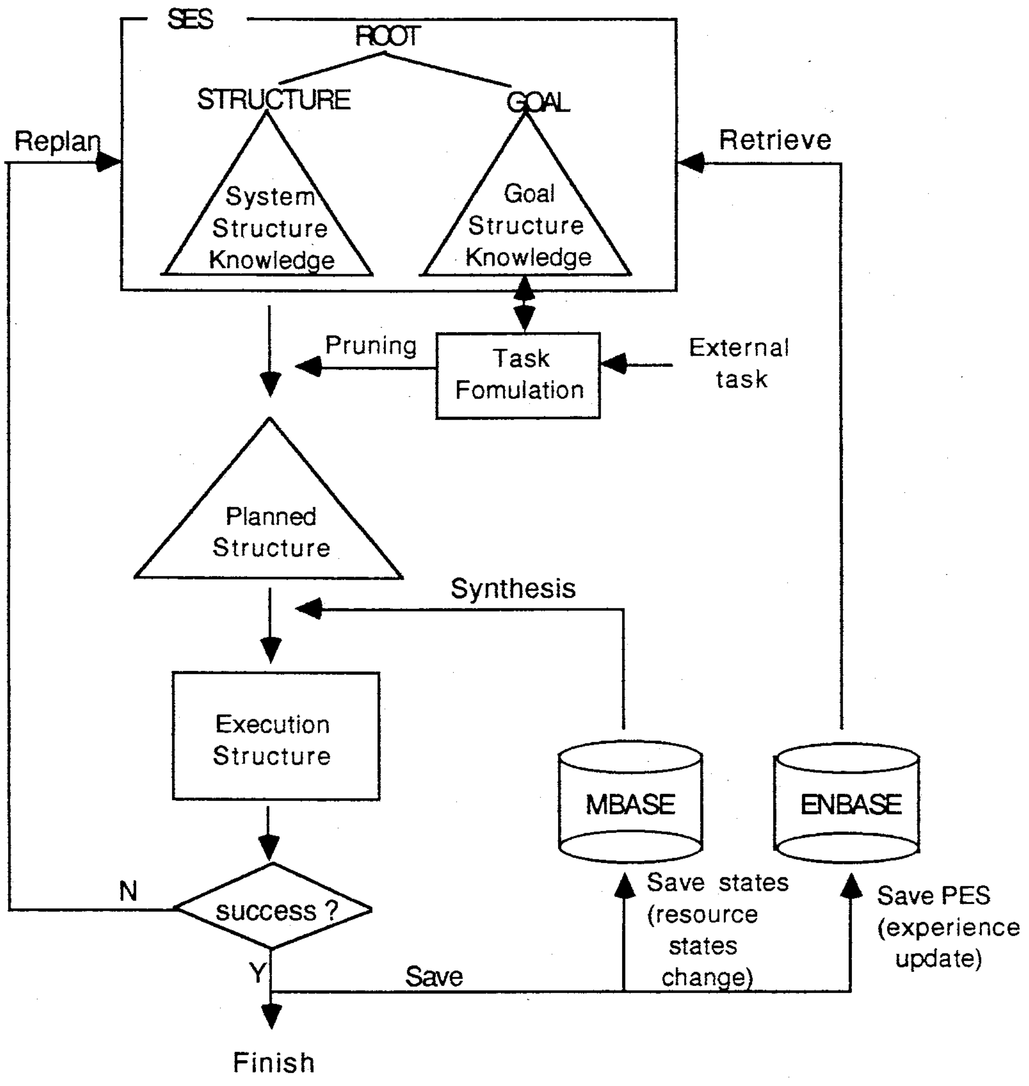

How to construct the system ?
System Structure Knowledge
- hard-form
- goal independent
- resources
- include control and diagnosis components
How to achieve goal ?
Goal Structure Knowledge
- soft-form
- goal driven
- skill (how to use resources)
- include planning and scheduling components
Pruning (Selection)
Planned Structure
Model synthesis
Execution

Figure 8.2. Planning Concept: Viewing Planning as Pruning the Goal Structure

In our SES/MB framework, planning is viewed as a pruning operation that generates a candidate structure. As shown in Figure 5.3 of Chapter 5, the STRUCTURE entity represents an execution structure that contains all possible configurations of a system. Any one of these configurations can be generated by pruning. On the other hand, the EXPERIMENT entity represents a goal structure that describes the possible alternative goals. The execution structure is pruned by the goal structure. After issuing a command, the system can autonomously interpret the command, plan an activation sequence, select models, execute the plan, and save the plan. Figure 8.3 depicts a model-based methodology for generating an autonomous system.



Where,
SES: System Entity Structure
PES: Pruned Entity Structure
MBASE: Model Base
ENBASE: Entity Structure Base

Figure 8.3. Autonomous System Generation Methodology

The overall procedures shown in Figure 8.3 are as follows:

1. Load the system entity structure of the space station laboratory (SSL) that organizes all available domain knowledge including experiment knowledge and structure knowledge.

2. Interpret natural-language-like task commands (objectives) in order to map them into the predefined domain knowledge (SSL).
3. Check whether there are already existing plans (pruned entity structures): if such structures exist (whole or partial PESs) use them, if not generate a sequence of actions by querying the goal knowledge part of the system entity structure (EXPERIMENT).
4. Construct a model structure by pruning the execution knowledge part of the system entity structure (STRUCTURE) according to the goal knowledge.
5. Transform the model structure into an autonomous system architecture by synthesizing component models from the model base, MBASE.
6. In the faulty case, reprune the SES after issuing the new goal and repeat from step 3.
7. Save the results for reuse; the states of each model into MBASE, and the PES into ENBASE.

8.5. Non-experienced Planning (Initial Planning)

The initial planning, where no previously experienced plans exist, can be achieved by a hierarchical search process employing a rule-based approach, in which the task sequence can be selected depending on goal commands and predefined constraints by checking pre-conditions and post-conditions of the state transition diagram.

The task formulation module interprets a natural-language-like command, which consists of a list containing the task name (i.e., the fluid operation name) and its variables (i.e., the liquid name and its amount), and use it to select subplanning models.

Let us consider the following external commands: (mix ((liquid-A 100ml) (liquid-B 100ml))) and (mix ((liquid-A 100ml) (liquid-B 200ml))). Suppose that three different sizes of syringes are available; small (≤ 100ml), medium (≤ 200ml), and large (> 200ml). In the case of the former external command, it suffices to employ one small syringe that can be used to handle both liquids. It is not necessary to put the syringe down in between and pick it up again. However in the latter case, two different syringes may be used: a small and a medium sized syringe. In that case, the robot needs to let go of one syringe and grab another. However, it would also be possible to use one medium sized syringe from the beginning and fill it only half with liquid-A, or we could use a small sized syringe throughout and fill it twice with liquid-B. Which task plan is best depends on other factors such as the price of disposing of non-reusable parts, the price of cleaning used syringes, the time allowed for task completion, or the accuracy available for filling syringes only half (maybe, no appropriate threshold sensors are available for such operation). In both cases, two types of liquids, liquid-A and liquid-B, are required.

Based on these pieces of information, that can be formulated in a rule base, the task formulation module can sequence its subgoals. The formulated goal command can be indexed and catalogued into a high level model (HM) for later reuse if the initial plan is executed successfully. Faulty command execution can be handled within the task formulation module using the feedback (replanning) loop. For this purpose, the task formulation module would have to consider not only external commands but also internal commands generated as a consequence of faults detected during execution of the initial or a previous task plan. However, this feature is not currently implemented yet.

Unlike the high level external (user) commands that cannot easily be categorized, most (medium level) fluid handling laboratory procedures consist of basically fixed sequences of common steps [hurst]. These building blocks are called middle level models (MMs). They link the external (user) commands to the low level dynamic resources. It is through some combination of these MMs that any laboratory procedure may be outlined. Each MM consists of subplans, so-called low level models (LMs). The following is an exemplary (partial) list of MM classes with their definitions and required tools [hurst].

Target MMs

1. Weighing (WEIGHING): Quantitative measurement of sample mass. The autonomous system must be capable of interfacing with common laboratory electronic scales, i.e.,

opening the scale door, taring a suitable container, delivering the desired amount of sample, and recording the actual weight of the sample.

2. Conditioning (MIXING, HEATING, COOLING): Modifying and controlling the sample environment. For example: shaker, heater, cooler.
3. Separation (SEPARATING): Coarse mechanical and precision separations. Several subclasses that fit into this category include: solid phase extraction, liquid/liquid extraction, precipitation, filtration, and centrifugation. For example: centrifuge, chromatograph, electrophoretic instrument.

Auxiliary MMs

1. Manipulation (OBJECT-GETTING, OBJECT-PUTTING): Physical handling of laboratory materials, e.g., the transfer of samples from a rack to the nozzle of a diluter station or to a sipper on a spectro-photometer. Pouring, capping, and crimping also fit into this category. For example: robot gripper.
2. Liquid-handling (SAMPLING, WASHING, FLUID-INJECTING, FLUID-ADDING, FLUID-EXTRACTING): All physical handling of liquids -- reagents and samples. The autonomy system must be able to handle and control the necessary stations to perform the desired pipetting, dispensing, and diluting called for in an application. For example: syringe, container.

As demonstrated in Figure 8.4, the action flow graph (state transition graph) implies some possible robot action flows in the fluid handling laboratory environment. Each MM has its cause-effect relation using an attribute called *pre-post-lst.* For example, the MM "fluid-injecting" has a *pre-post-lst* attribute ((sampling heating) (sampling washing)). That is, if the previous action was "sampling," then the next action should be either "heating" or "washing." The MM sequence depends on the target MM and other system requirements such as the number of syringes in use. Once a target MM and necessary instruments are selected, a proper sequence can be determined by chaining MMs in backward and forward manner starting from the target MM. Conflict resolution can be accomplished within the task formulation module. For example, if "mixing" MM and one syringe are selected as a target MM and system requirement, respectively, then the resultant sequence are: “object-getting,” “sampling,” “fluid-injecting,” “washing,” “sampling,” “fluid-adding,” “mixing,” “washing,” “object-putting,” and “finishing.” On the other hand, if two syringes are selected, then the resultant sequence are: “object-getting,” “sampling,” “fluid-injecting,” “washing,” “object-putting,” “object-getting,” “sampling,” “fluid-adding,” “mixing,” “washing,” “object-putting,” and “finishing.” With this MM sequence and other specified resources/constraints such as the number of required instruments and their sizes, the planning system can prune the STRUCTURE into a desired execution structure. This pruned entity structure is then transformed to the simulation environment by synthesizing necessary models from the model base.

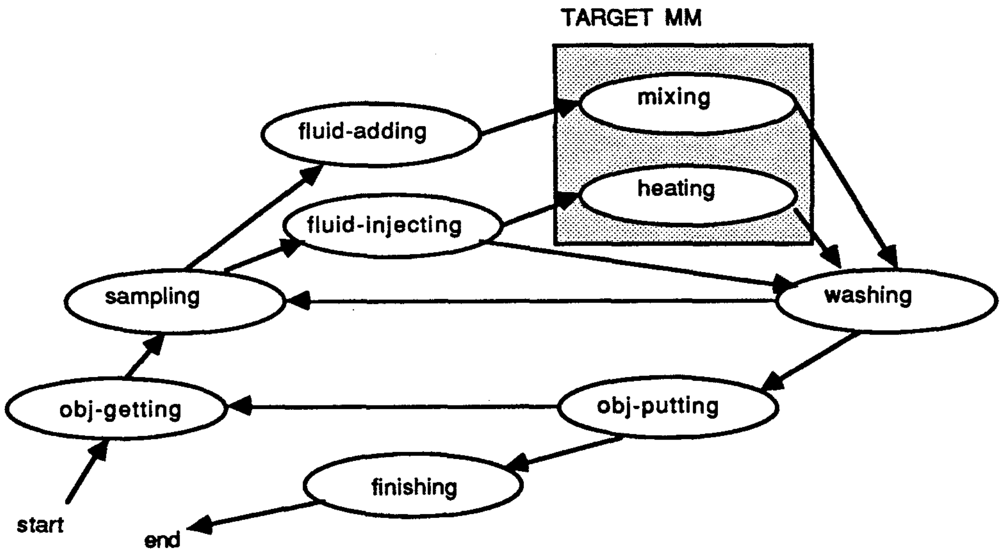


Figure 8.4. Action Flow Diagram between MMs

The MM is again decomposed into a sequence of low level models (LMs) specified at a higher resolution. For example, the “sampling” MM consists of four LMs sequence: “MOVE,” “PLUG,” “SYRINGE,” and “PLUG,” with the subgoals: *storage-A, push, fill-it,* and *pull*, respectively. In this way, high level goals and constraints can be hierarchically mapped and integrated with low level dynamic models. Each LM can be developed using the goal table of the internal operational model (see Figure 7.8(b)) in the LMPU as discussed in Chapter 7.

8.6. Experienced Planning

As discussed in Chapter 3, more than one SES may exist in the entity structure base (ENBASE), each one representing a family of models for its root entity. In principle, every entity might have its own SES but this would lead to extreme fragmentation of the encoded knowledge. Reasons for giving an entity its own SES include: the large size of its family of possible prunings, its high likelihood of being modified, and its occurrence in several places of existing SESs [109]. Figure 8.5 represents a partitioned SES and its indexed PES family for reuse. The partitioned PES is crucial to achieve dynamic planning (retrieve and update old plans).

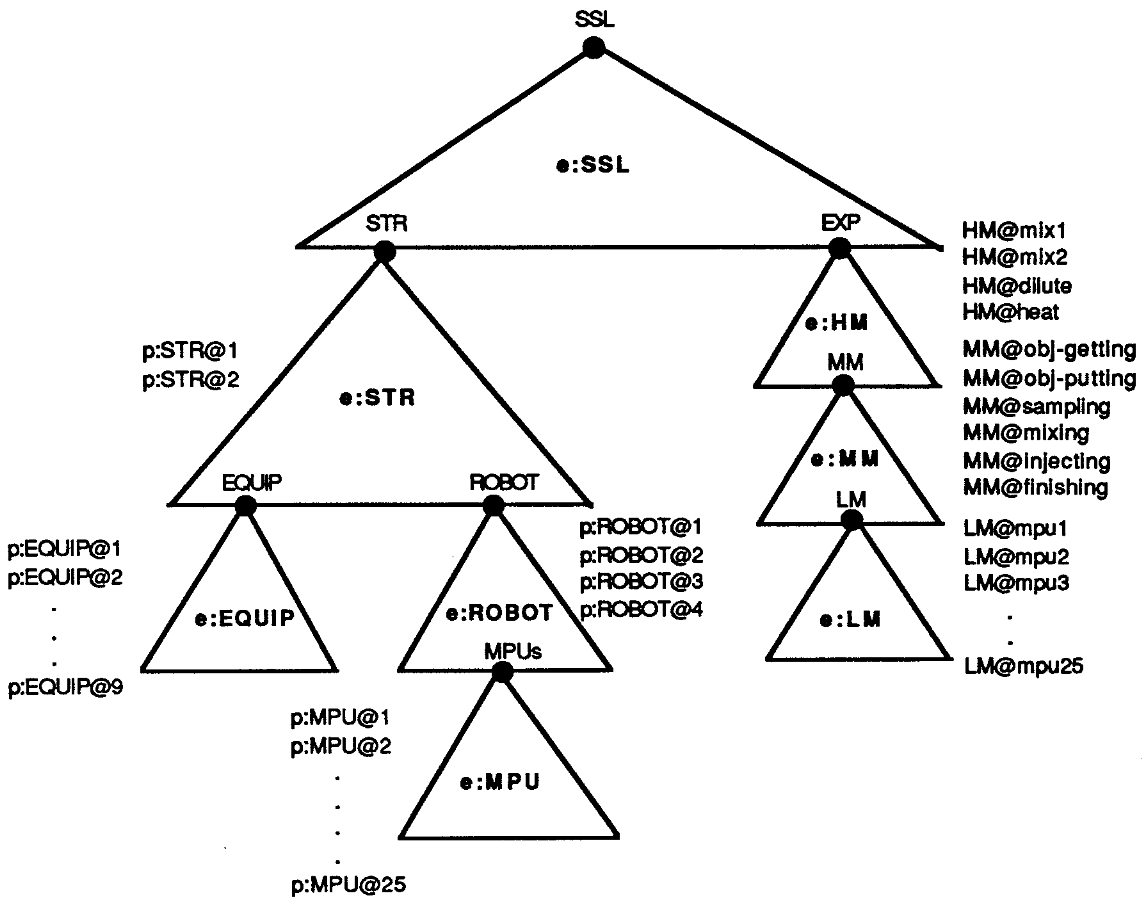


Figure 8.5. Hierarchical Structure of Partitioned/indexed PES family
in Figure 5.3 for reuse

For example, suppose we already have experience with the "HEAT" but not the "MIX1" HM. As presented in Figure 8.4, the "MIX1" HM requires the "fluid-adding" and "mixing" MMs in addition to the MMs used in the "HEAT" HM except for the "heating" MM. Thus, we can develop a new "MIX1" HM by using previously generated partial plans from the "HEAT" HM and by generating new plans for the "fluid-adding" and "mixing" MMs.

In the following sections, we describe how the planning and execution structures are coherently integrated as an extended planning paradigm using the abstraction related models on the execution hierarchy corresponding to hierarchical planning models.

8.7. Abstraction-Related Modelling

Once a plan is generated, the execution hierarchy is constructed in accordance with the plan hierarchy. A model for event-based control discussed in Chapter 7 must represent not only a decision making component, but also the model of the real system, which is used by a decision making component to arrive at its decisions. Such abstraction is based on homomorphic preservation of the equipment input-output behavior where inputs are operation commands to the equipment and outputs are responses of finite state sensors attached to the equipment to observe its state. Selection of controls and sensors must reflect the operation objectives. An atomic DEVS model abstracts incremental micro-state transitions from the continuous model and replaces them by time windows taken for macro-state transitions (which correspond to crossing of sensor thresholds) [55],[56]. A coupled DEVS model similarly abstracts the behavior of the composition of lower level DEVS models.

Figure 8.6 represents abstraction related models. At the bottom, MB is a continuous state model of the system being controlled (the most refined model considered for it). Other models are related by abstraction, i.e., some form of homomorphic relation. ME is a discrete event model derived from MB whereas LMI-O and LMI-D are two different abstraction models of MI (for operation and diagnosis, respectively) which in turn is an abstraction of ME. Each abstraction is governed by an underlying morphism. ME serves as the external model of each device whereas LMI-O and LMI-D serve as the internal models of the lowest level event-based control unit (LMPU).

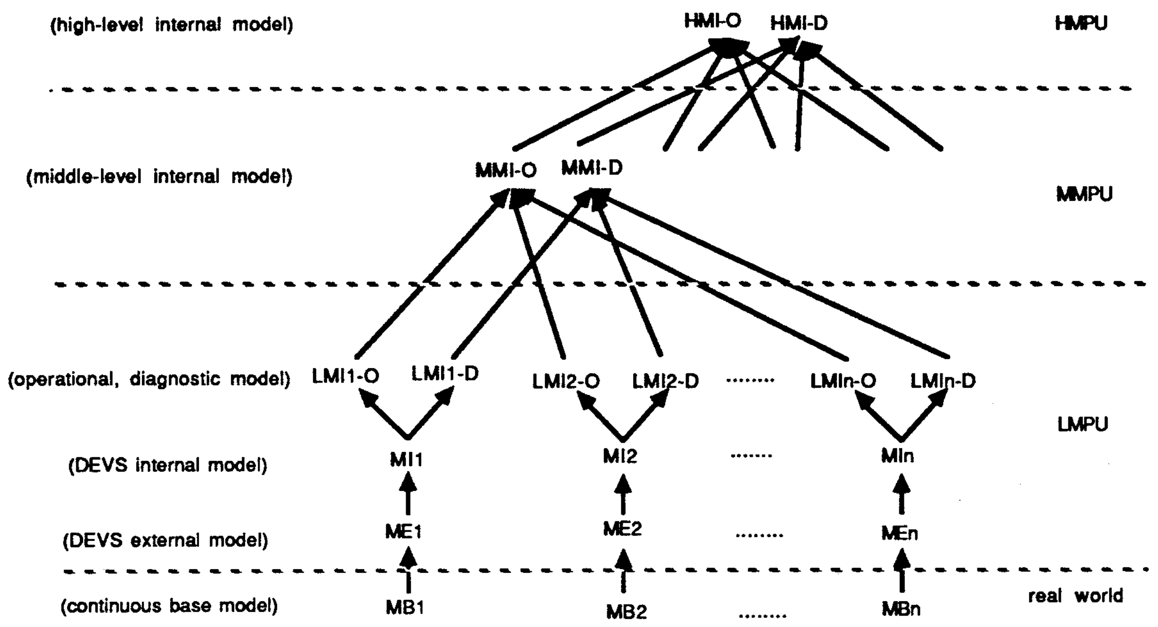


Figure 8.6. Abstraction Related Models

Sets of LMI-Os and LMI-Ds can be composed into higher level models (MMI-Os and MMI-Ds) and abstracted to represent more global state transitions (used by MMPU). Likewise, the HMI-Os and HMI-Ds are the highest level models, which represent the global state transitions (used by HMPU). In this way, the higher level execution elements use their own models, which are abstractions of compositions of lower level models, to control their subordinates. The abstraction process of the operational models is achieved by aggregating states and windows of lower level models. The abstraction process of the diagnostic models is discussed in the following chapter.

Figure 8.7 uses a "sampling" MMI-O example in our robot-managed laboratory to depict the abstraction relation between different levels, where segmented sub-states and time windows of lower levels are aggregated/abstracted from a bottom-up viewpoint. To perform a fluid sampling unit action in a space-borne laboratory, a robot should act according to the following steps: 1) move to the target storage, 2) push a syringe in the storage receptacle, 3) wait for the liquid to fill the syringe, and 4) pull out the syringe. Such action sequences (state transitions) can be abstracted by retaining only the global state transitions achieved by lower level units. As shown in Figure 8.7, the robot starts from an initial location with an empty syringe and ends up at the location of storage-A with a full syringe. The global time consumed and its window are an appropriate summation of the time windows of the component actions. While the summation of the minimum times is always appropriate, summation of the time windows may lead to overly conservative assumptions about the system behavior. Yet, this is a very simple and straightforward approach that was used in our work.

The levels of abstraction just illustrated can be formalized using system morphism concepts [56],[108],[109]. By generating a hierarchical structure with models of different levels of abstraction we can systematically integrate highest level goal commands with the most refined dynamic models.

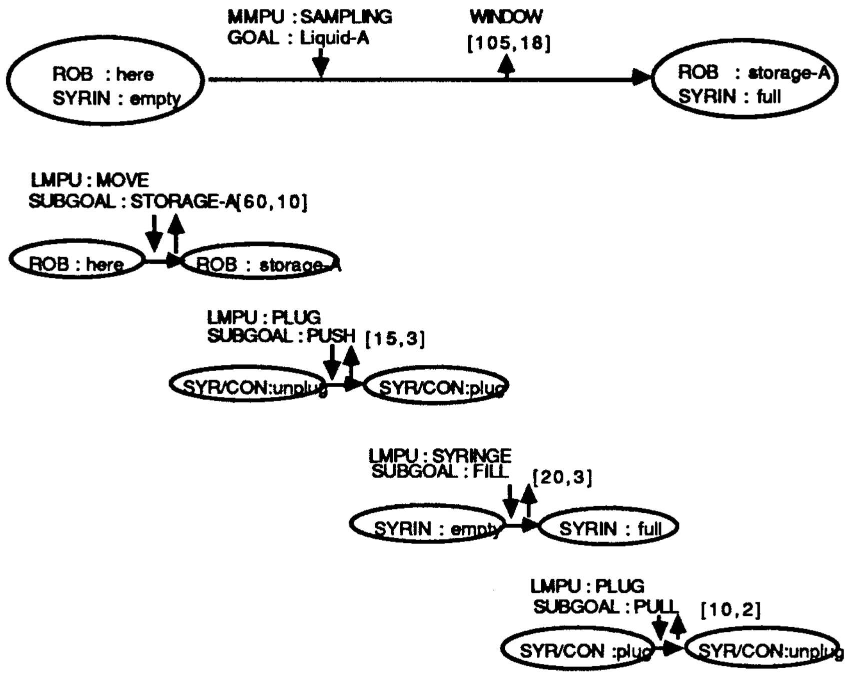


Figure 8.7. Abstraction Process from LMI-Os to MMI-O: sampling example

8.8. Hierarchical Planning/Execution Structure

The robot's cognition system (BRAIN) exemplifies *a hierarchical planning / execution structure* as shown in Figure 8.8. Once a plan is generated by using EXPERIMENT knowledge (i.e., HMs, MMs, and LMs) in a top-down manner, corresponding internal models for MPUs can be retrieved from the model base or automatically generated by using the abstraction process in a bottom-up manner, and by attaching a controller to each model, the hierarchical execution structure can be constructed.

At each level, the control unit has its own internal model and controller to supervise its sub-component units. There are two types of messages in the hierarchy: the *goal* (command) and *done* (response) messages. The *goal* is divided into a set of subgoals in a top-down manner, whereas, the *done* messages are gathered in a bottom-up manner. There are three types (+,0,-) of *done* messages: + stands for *success*, - represents *failure,* and 0 denotes *undecisiveness.* The last message may be due to a lack of relevant sensors or a complex fault associated with other units in the hierarchy, a topic being discussed in the following chapter.

As a concrete illustration, let us set up a mixing experiment: *"mix x amount of liquid-A with y amount of liquid-B"* . To perform such a task, a robot must identify a syringe required to sample a liquid-A, then bring that syringe to the identified storage in which liquid-A is stored, perform the sampling from the storage, and so on. As presented in Figure 8.9, the HMPU manages such activation sequences (MMPUs), whereby each action unit (MMPU) is further divided into yet smaller dynamic model units (LMPUs).

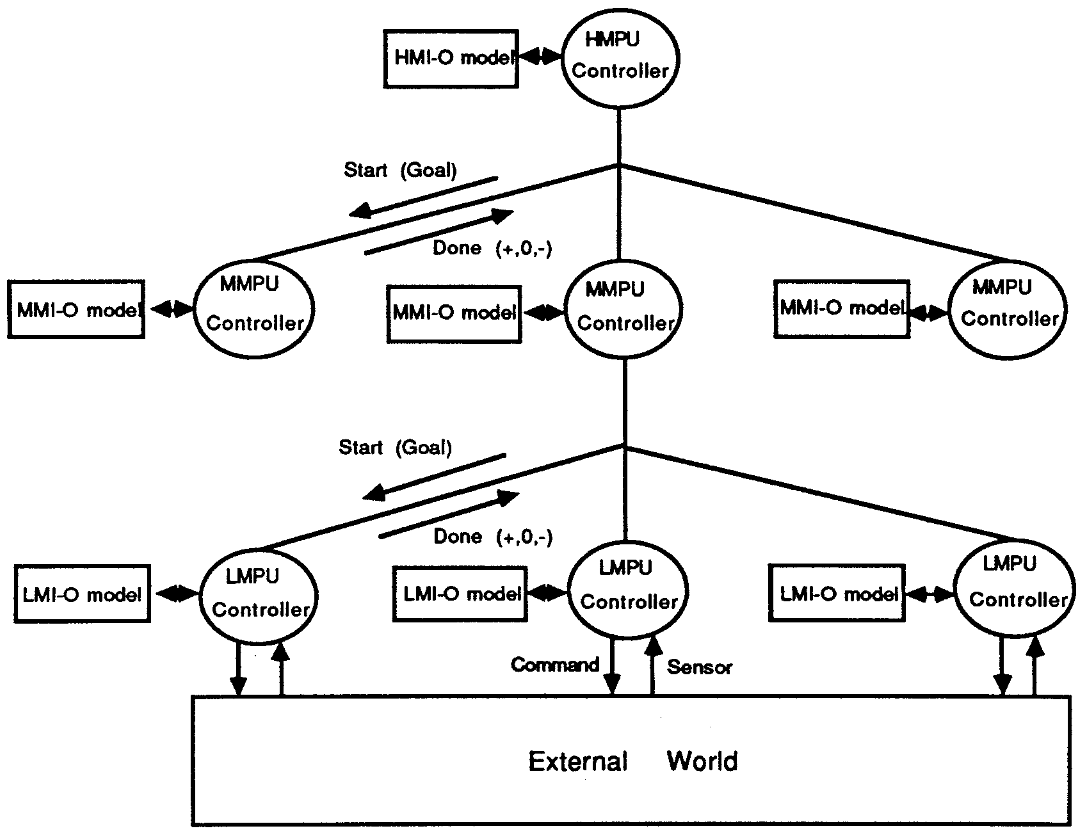


Figure 8.8. Hierarchical Planning/Execution Structure

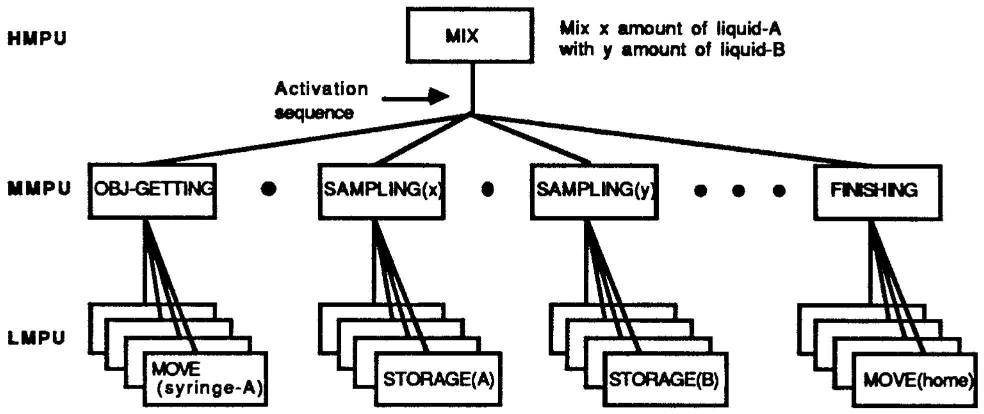


Figure 8.9. Mixing Example of Hierarchical Structure

8. 9. Conclusions

The key concepts and issues of an extended planning paradigm including execution (simulation) are categorized as follows:

1. Abstraction-related hierarchical planning.
2. Reusability of old plans.
3. Time constraints (goal condition checking).
4. Replanning.

In this chapter, we have described how the above concepts can be formulated and integrated within the model-based planning approach. Building on the basis of the SES/MB framework, we have extended the ability of our knowledge/model base tools to support hierarchical model-based planning system design through the ability to generate families of planning alternatives, reusability of models, and model base coherence and evolvability.

CHAPTER 9
HIERARCHICAL MODEL-BASED DIAGNOSIS

9.1 Overview

The diagnostic procedure presented in this chapter is model-based, inferring single or multiple faults from the knowledge of faulty behavior of component models and their causal structure.

The overall goal of this chapter is to develop a hierarchical diagnostic system that exploits knowledge of structure and behavior. We have proceeded by building a software environment that uses such knowledge to reason about faults using advanced simulation modelling techniques. The main focus is on model-based diagnostic capabilities of high autonomy systems. We have implemented such a system based on a number of new ideas and tools using the SES/MB framework.

We use two types of diagnosers, local and global diagnosers, to find single or multiple faults using knowledge of structure and behavior. By local diagnosis we mean the diagnostic description of a component model: once a symptom is detected, the local diagnoser can discover a fault that has occurred within the currently active model unit. In contrast, by global diagnosis we mean the diagnostic description of a coupled model [16],[100]. The global diagnoser uses a symbolic simulation as discussed in Chapter 4 to generate all possible causal trajectories of its component models.

This chapter is organized as follows. First it reviews the basic concepts of local and global diagnosis. This discussion is followed by a hierarchical diagnosis of an autonomous system that has been implemented for our space-borne laboratory.

9.2. Background of Shallow and Deep Reasoning

The diagnostic process is defined as the recognition of diseases or faults and their causes. A diagnoser remains a passive observer of its system until it detects an anomaly. For this purpose, the diagnoser assumes that its model of the system is correct, and looks for discrepancies between the behavior of its internal model and the behavior of the real system. If such a discrepancy has been detected, it assumes that the system has undergone some change, which is considered a fault or disease.

Once an anomaly has been detected, the diagnoser switches from a passive to an active mode. Its next task must be to come up with a set of hypotheses as to what might have gone wrong. It can do this by shallow reasoning, i.e., using a rule-based approach. In this scenario, the diagnoser will traverse a cause-effect matrix in reverse direction. The anomaly is treated as an effect (symptom), and the diagnoser checks which causes could possibly be made responsible for the observed symptom. Alternatively, it can do this by deep reasoning, i.e., using a model-based approach. In this scenario, the diagnoser will make use of fault models to match the observed anomalous behavior of the system. Each model can then be used to generate one or several hypotheses. Shallow reasoning has the advantages of being generally faster and of aiming directly at potential causes of the observed anomaly, while deep reasoning uses an indirect approach to determine potential causes. Deep reasoning has the advantage of being theoretically able to recognize even unforeseen causes, whereas shallow reasoning cannot hypothesize beyond the scope of the previously determined set of potential problems, which are hard-coded into the cause-effect table.

Shallow reasoners find single faults usually faster than deep reasoners. However, the fault tree (a refined hierarchical version of the cause-effect matrix) grows exponentially with the number of simultaneous faults being considered. This is not the case with deep reasoners, although computation time may grow in the same manner. Also, deep reasoners are much less vulnerable to missing data than shallow reasoners. They are therefore more robust. Consequently, both types of reasoners have their place, and a hierarchical combination of the two may often be the best solution [80].

In our model-based diagnosis approach, shallow and deep reasoning concepts are coherently developed and integrated in the form of a hierarchical model structure for local and global diagnosis, respectively. To do this, we attach diagnostic engines and associated models to the nodes of the task execution hierarchy discussed in Chapter 8. As illustrated in Figure 9.1 (see also Figure 8.8), there are two types of nodes, leaves and higher level nodes, which require local and global diagnosers, respectively. A local diagnoser detects faults caused within a monolithic model (action) unit, whereas global diagnosers detect faults propagated through an execution sequence.

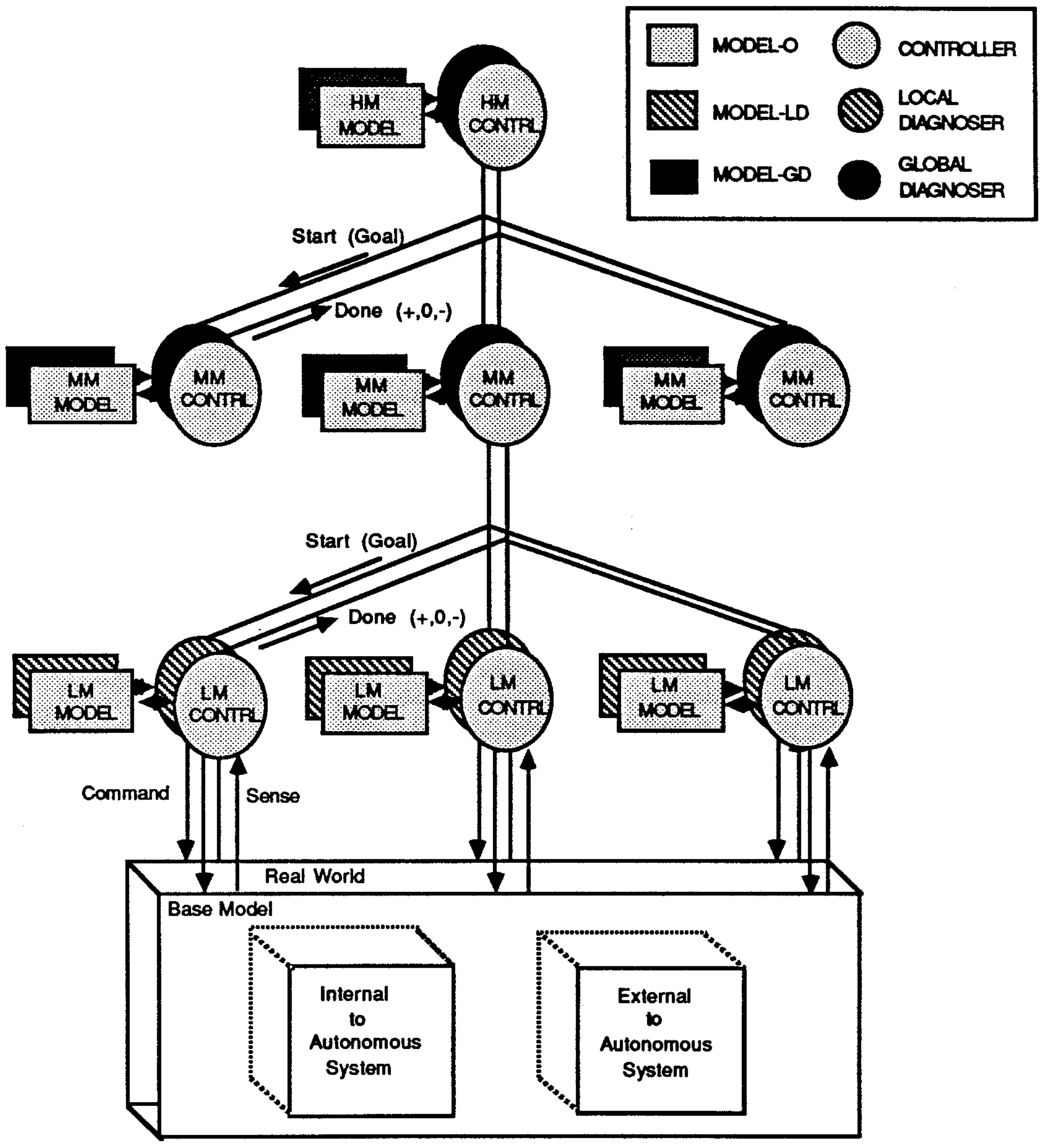


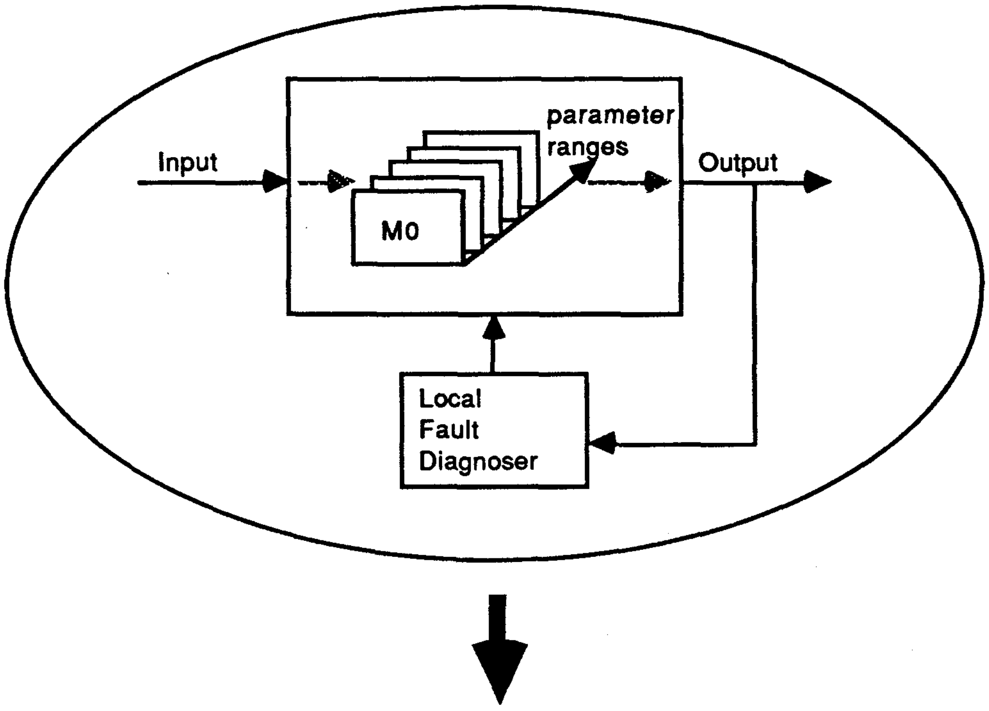
Figure 9.1. Hierarchical Execution/Diagnosis Structure

9.3. Local Diagnosis

By local diagnosis we mean the diagnostic description of a component model: once a symptom has been detected, the local diagnoser can discover a fault that has occurred within the currently activating model unit.

In the event-based control logic, the operator and diagnoser are directly coupled so that, once the operator has detected a sensor response discrepancy, the local diagnoser is activated. Data associated with the discrepancy, such as the phase in which it occurred and its timing (e.g., *too-early, too-late*), are also passed on to the local diagnoser. From such data, as well as information it can gather from auxiliary sensors, the local diagnoser tries to discover the fault that has occurred.

The local diagnostic model employed by the local diagnoser can be derived from the external model. Indeed, the local diagnostic model is an inversion process going from external effects to underlying causes. Figure 9.2 represents a local diagnostic model generation where the cause-effect table can be directly obtained by associating each model state depending on its parameter range (potential cause) with its resulting output (expected symptom). The validity criterion for the local diagnostic model is thus the following: the occurrence of every fault representable in the external model should be identifiable by the local diagnostic model, under the operation of the diagnosing inference engine. The inference engine implements a basic forward chaining approach in which all diagnostic sensor readings are requested first, and afterwards the diagnostic model is exercised until a diagnosis is made or all rules are fired without a conclusion being reached. A more detailed description of local diagnosis was given in Chapter 7.



Output (Effect)	Model (Cause)
O1	M1
O2	M2
O3	M3

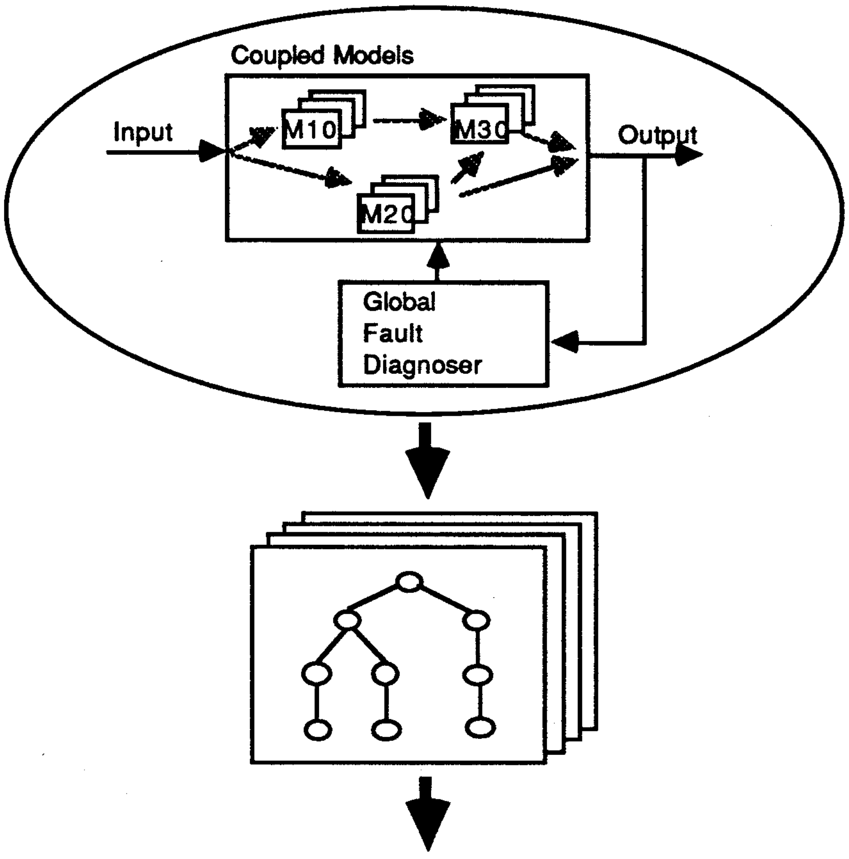
Where M_0 = normal model and M_j ($j > 0$) = fault model

Figure 9.2. Local Diagnostic Model Generation

9.2.2. Global Diagnosis

Our approach to global diagnosis is compatible with Reiter's *General Theory of Diagnosis from First Principles* [71] based on earlier work by de Kleer and Williams [25]. Intuitively, a diagnosis is a conjecture that a set of components of a system are faulty and the rest are normal. Reiter invokes a principle of parsimony to restrict attention to the subfamily of minimal diagnoses. However, Zeigler [100] shows that non-minimal diagnoses may exist where information is available to restrict the class of possible faulty versions of components and that the diagnosis process is severely impaired by not taking such non-minimal diagnoses into account. Thus the full family of diagnoses, although potentially much more computationally demanding than the set of minimal diagnoses, should be taken as the basis for trouble shooting. Our approach to global diagnosis generates such a full family of diagnoses.

The global diagnostic engine is activated by receiving a symptom (the detected anomaly). It activates a symbolic simulation to generate all possible trajectories, and marks those that reach states exhibiting the detected symptom. Alternatively for small models, such trajectories can be pre-compiled off-line for faster on-line use. Each such trajectory represents a hypothetical sequence of fault injections and activations that could have resulted in the observed symptom. The diagnoser investigates each trajectory by testing components in which the faults were injected for presence of such faults. Additional sensory probes may be used for this purpose. A trajectory for which all injected faults are found to exist is confirmed as an acceptable explanation of how the anomaly may have been caused. Trajectories are investigated in priority order based on the number of faults involved -- singletons first, pairs next, etc. Figure 9.3 shows the global diagnostic model generation where symbolic simulation is employed to generate every possible causal trajectory of its component models.



Output (effect)	Models (causes)
O1	M11 M32
O2	M22

Where $M_{i,0}$ = normal model and $M_{i,j}$ ($j > 0$) = fault model

Figure 9.3. Global Diagnostic Model Generation

As a concrete example of the interaction between local and global diagnosers, consider that our desired plan is to sample a small amount of liquid-A from storage-A. To achieve this, a robot should act in the following sequence: move to the syringe desk, pick up the small syringe, move to storage-A, plug the syringe into storage-A, fill the liquid-A into the syringe, and so on. However, suppose a robot picks up a large syringe instead of a small syringe for some reason such as "arm motor abnormal" or "mobile motor abnormal," etc. If such an abnormal condition is not out of range of its action unit, then such an abnormality may be propagated to the filling action where the timing should not be on time, i.e., *too-late*, because the actual syringe is large while the robot considers it to be small. In such a situation, the local diagnoser for the syringe is activated. As described in Chapter 7, the local diagnoser uses two types of sensory information: *indicator* (expected sensor, timing, phase) and *vision* (level, angle, constriction). In our example, the sensory information will look as follows: sensor = *full-sensor*, timing = *too-late*, phase = *filling*, angle = *normal* (i.e., $89 < \text{angle} < 91$), constriction = *normal* (i.e., $\text{constriction} < 10$), and $\text{level} \geq 100$ (where 100 is the full level of a small syringe). Therefore the local diagnostic conclusion will be "*don't know*" as shown in Figure 9.4. Consequently, the global diagnoser will be activated.

indicator			vision			conclusion
timing	sensor	phase	angle	constriction	level	
too -late	*full -sensor*	*filling*	*normal*	*normal*	*≥ 100*	*don't know*
					< 100	*leaky*
				constricted	*don't care*	*constricted*
			off	*normal*	*don't care*	*off angle*
				constricted	*don't care*	*constricted or off angle*

Figure 9.4. Sensory Data and Diagnostic Conclusion for Local Diagnoser (partially shown)

Once the global diagnoser has been activated, it first builds a simulation environment, i.e., it collects its children models and attaches corresponding fault generators, and then simulates to discover the actual faults. If this unfortunately fails again, which means that the cause of the fault must be propagated further from the previous action unit, a yet higher level diagnoser is being activated that will use the previously active lower-level global diagnoser as well as other lower-level diagnosers to check faults associated with previously as well as newly activated models.

Figure 9.5 shows the concept of our fault modeling approach. The injected fault from the fault generator to the fault model changes the world status and propagates it to the next model. When the world status value reaches its detection boundaries, the model is passivated in a faulty state (effect).

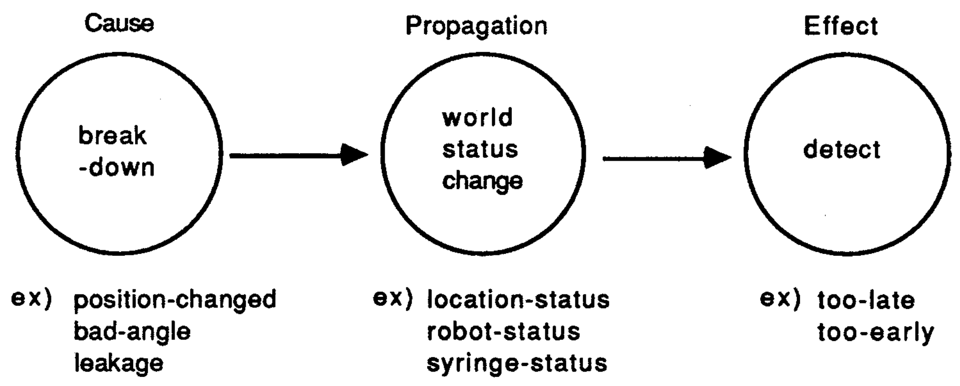
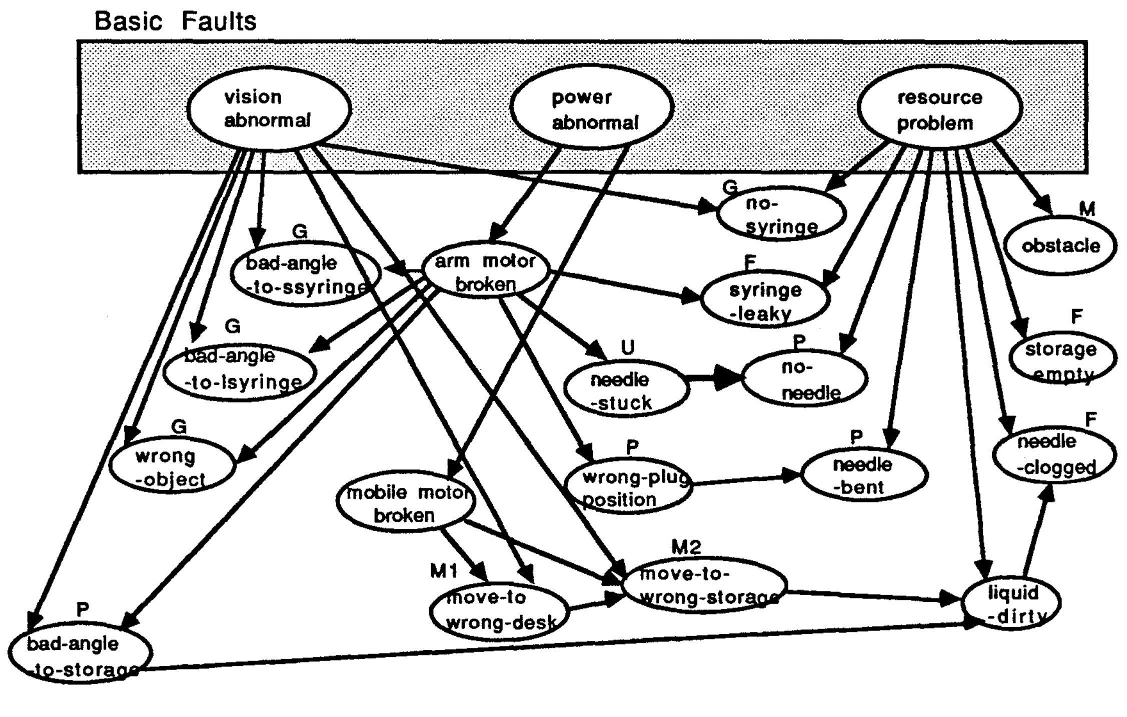

Cause
Propagation
Effect
break -down
world status change
detect
ex) position-changed bad-angle leakage
ex) location-status robot-status syringe-status
ex) too-late too-early

Figure 9.5. Causal Propagation Modeling Approach

Fault generators and fault models have symbolic times so that a symbolic simulation of a network of such components will generate all state trajectories consistent with such a set of imposed constraints. Four types of constraints on faults can be considered:

1. State dependent faults: once a model state is entered, a fault should be activated; conversely, when a state has been left, a fault should be de-activated.
2. Causally dependent faults: once a fault is activated, another fault should be activated.
3. Mutually exclusive faults: once a fault is activated, another fault cannot be activated.
4. Timing related faults: a fault is to be activated before another fault is activated.

Figure 9.6 illustrates the causal relation between faults of our space-borne laboratory application, for example, the faults "bad-angle-to-ssyringe" and "bad-angle-to-lsyringe" have mutually exclusive characteristics. Those two faults obviously cannot co-exist. The faults "move-to-wrong-desk" and "move-to-wrong-storage" could have a causally dependent relation because both may be caused by the same motor abnormality. The fault "bad-angle-to-storage" should precede the fault "needle-bent" when both faults co-exist, because the needle will be bent only as a consequence of plugging in the syringe at an odd angle.



Where M1: Move1, G: Getting, M2: Move2, P: Plug, F: Filling, and U: Unplug model unit

Figure 9.6. Causal Diagram of Fault Family

Figure 9.7 shows how components of a multicomponent model are modelled for fault propagation analysis. Each component is abstracted into a unit consisting of one, or more, fault generators and a fault model. An anomalous condition is sent by a fault generator to the fault model that changes its state accordingly. Activation and de-activation of fault generators is accomplished by appropriately coupling the output ports of its influencing components to its activate/de-activate input ports. Timing constraints on fault generation are supplied as initial elements of the constraint sets that are built up along the trajectories. Such explicit representation of causal relations between multiple faults and/or fault models strongly support the analysis of complex structural diagnostic problems.

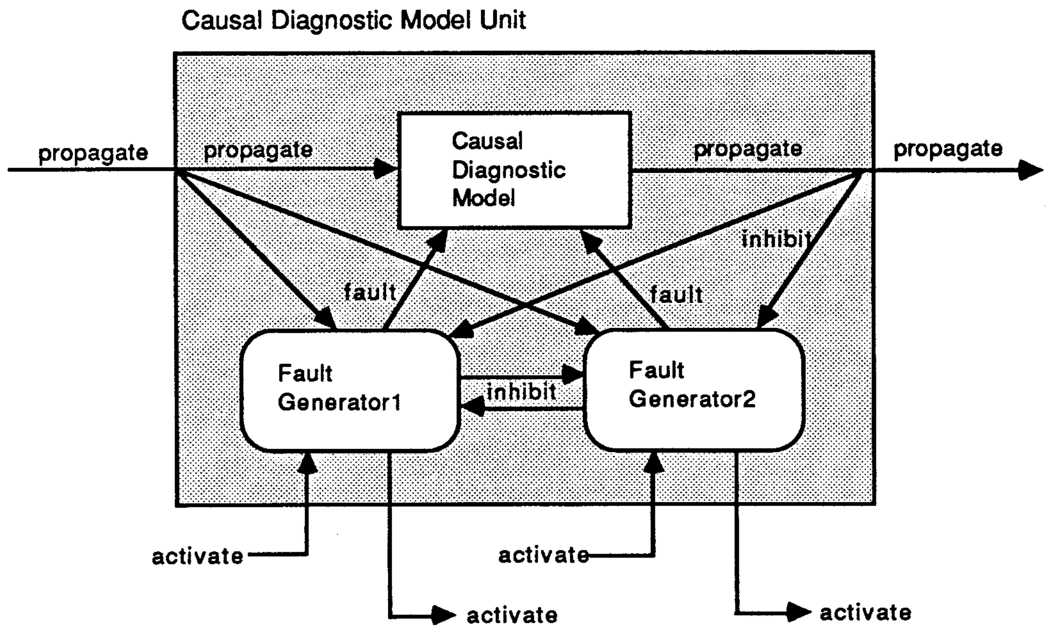


Figure 9.7. Causal Structure of Diagnostic Model Unit

Using pseudo-code, the causal diagnostic model can be described as:

```
when receive world-status on "propagate" or "fault" port,
        if world-status values are within detectable range,
        then change its state into faulty state and passivate,
        else, propagate the previous world-status to next model,
        and "inhibit" the activation of latent fault generator related to
        previously activated fault model.
```

The fault generator can be described in pseudo-code as:

```
when receive "propagate,"
        wait a given time (fault injecting time) and send "fault" to its diagnostic model,
        and if there exist causally dependent faults, then send "activate" to them,
        if there exist mutually exclusive faults, then send "inhibit" to them.

when receive "activate,"
        set fault injecting time < model activation time.

when receive "inhibit,"
        passivate.
```

Figure 9.8 presents a multicomponent model for event-based execution of an elementary laboratory task. The MOVE component represents a robot travelling from one location to another. An associated fault generator generates the occurrence of an obstacle in the way. The fault model abstraction for MOVE should represent the effect of the obstacle on its state -- perhaps to lengthen its time to reach its destination. The effect of the fault on other components is modelled by passing a world status description along as output from a component to its influencees. For example, if the obstacle causes the MOVE component to arrive at its destination slightly askew, this departure from the normal state will be passed on in the world status description to subsequent fault model components. Fault models include transitions to faulty (dead) states that represent detectable anomalous conditions. An anomalous condition is detectable at a fault model component if there is a sensor affected by this condition in the corresponding operational model and a time window in which the sensor is expected to respond. For example, a slight delay in arrival at destination at a skewed orientation may not be detectable by the MOVE component itself. However, the skewed orientation may cause the robot to select the wrong type of syringe and this might be detected subsequently by the SYRINGE model due to a mismatch in filling time.

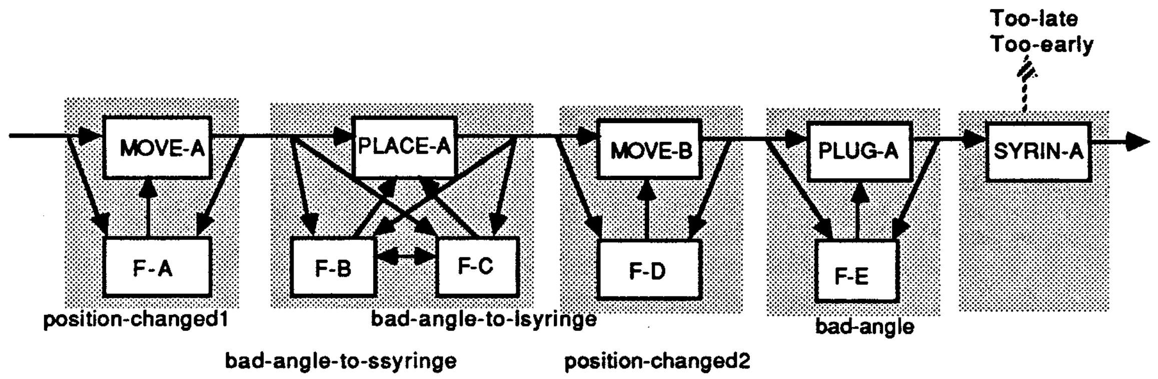


Figure 9.8. Causal Propagation Model Structure: Sampling Example

In the mixing example shown above, the symptom *too-early* in model SYRIN-A can be caused by a combination of two faults: “position-changed1” and “bad-angle-to-ssyringe”. Similarly, the symptom *too-late* in model SYRIN-A can also be caused by a combination of “position-changed1” and “bad-angle-to-lsyringe” or by a combination of “position-changed2” and “bad-angle”, or finally by a mix of both combinations.

As illustrated in Figure 9.9, the searching space can be greatly reduced by adding constraints between faults discussed earlier. By specifying a mutually exclusive relation between F-B (bad-angle-to-ssyringe) and F-C (bad-angle-to-lsyringe) and a causal dependency relation between F-A (position-changed1) and F-B or F-C, and also between F-D (position-changed2) and F-E (bad-angle), we obtain the relatively simple tree structure shown in Figure 9.10. From the generated trajectories, we finally get the cause-effect table. For example, in the *too-early* case, the table shows that the robot position first obtains a wrong value (fault) during its move toward the syringe-A, which then causes the

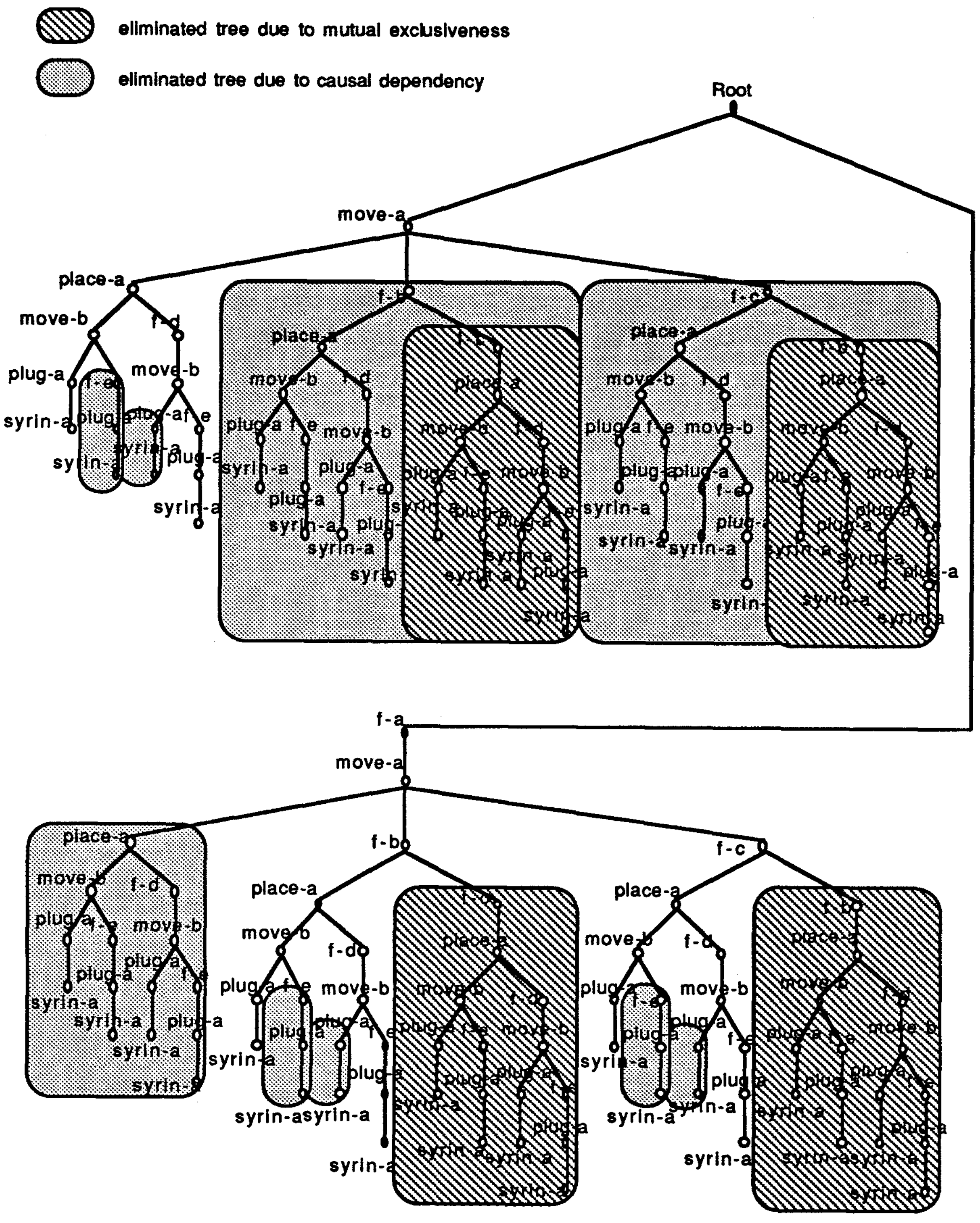
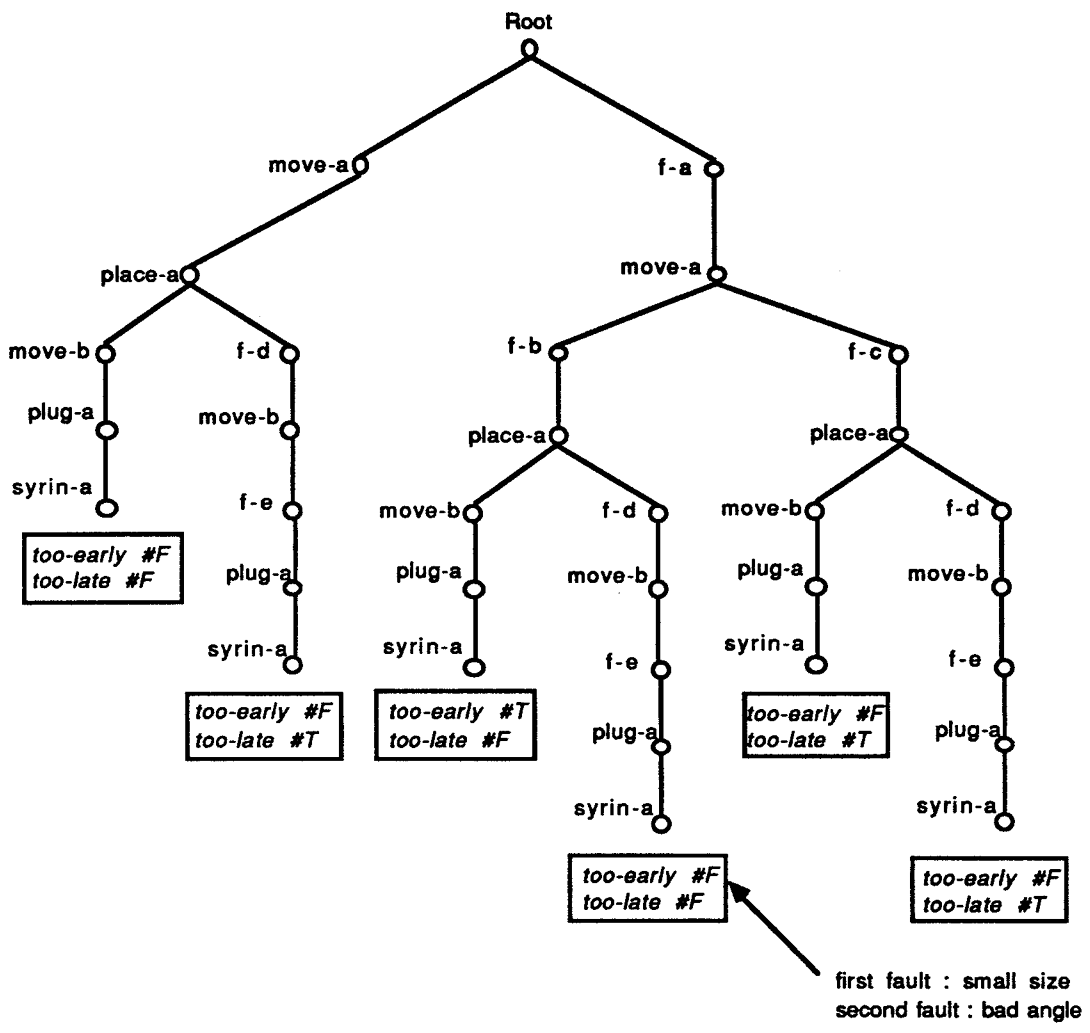

eliminated tree due to mutual exclusiveness
eliminated tree due to causal dependency
Root
move-a
place-a
move-b
f-d
plug-a
syrin-a
f-b
f-c
f-e
f-a

Figure 9.9. Tree of Trajectories of Example in Figure 9.8.



Detected Anomaly	Fault Trajectories
SYRIN-A in too-early	((MOVE-A position-changed1) (PLACE-A bad-angle-to-ssyringe) (MOVE-B ()) (PLUG-A ()))
SYRIN-A in too-late	((MOVE-A ()) (PLACE-A ()) (MOVE-B position-changed2) (PLUG-A bad-angle)) ((MOVE-A position-changed1) (PLACE-A bad-angle-to-lsyringe) (MOVE-B ()) (PLUG-A ())) ((MOVE-A position-changed1) (PLACE-A bad-angle-to-lsyringe) (MOVE-B position-changed2) (PLUG-A bad-angle))

Figure 9.10. Trajectory Tree with Fault Constraints (state dependant, mutually exclusive, causally dependant) and Its Diagnosis Table of Example in Figure 9.8.

robot to pick up the small syringe, which is smaller than its expected size (subsequent fault) so that finally the full-sensor of syringe-A responses faster than expected (registered symptom). Notice the special case in which a too small syringe was injected at an odd angle. The effects of the two faults cancel each other out, and therefore, this fault combination goes undetected.

9.5. Conclusions

A hierarchical model-based diagnosis system has been developed. Local diagnosers can diagnose faults whose effects occur within the same component as the detected discrepancy. Faults whose effects are propagated to other components are not locally diagnosable. To diagnose such faults we can use a global diagnostic engine and its associated model class. To do this, we adopted a symbolic simulation technique stemming from linear programming to support automatic generation of all trajectories that are consistent with a detected operation anomaly. In contrast to Reiter's theory [71], we represent faults occurring in time, and analyze how systems respond to fault occurrences.

This research has made several contributions: First, this approach to model-based diagnosis supersedes other approaches in that it intrinsically represents timing effects. Second, the system is able to diagnose failures related to multiple faults. Third, it can discover all fault models as well as injected fault types that are consistent with the observed faulty behavior. Fourth, a clear separation is drawn between diagnostic engines and diagnostic models, resulting in a domain (and inference procedure) independent diagnostic procedure. Fifth, a separation of local and global diagnosers ensures a proper mapping onto the hierarchical nature of our intelligent system design, resulting in efficient diagnostic procedures. Sixth, the system is developed under the SES/MB modeling and simulation environment, resulting in a direct cooperation with other intelligent units such as the task planner, the event-based controller, and the repairer (replanner).

CHAPTER 10

MODEL-BASED ARCHITECTURE FOR HIGH AUTONOMY SYSTEMS

10.1 Overview

To cope with complex objectives, an autonomous system requires integration of symbolic and numeric data, qualitative and quantitative information, reasoning and computation, robustness and refinement, discrete event system specification and continuous system specification.

In general, the AI approach is too qualitatively oriented to handle quantitative information very well. For example, classic AI planning approaches [65],[76],[86] do not consider the timing effects, which should be of primary concern in representing our dynamic world. On the other hand, control researchers have a fairly narrow view-point, so that they mainly focus on refinement rather than robustness of a system [89], and they usually consider only the normal operational aspects of a system [54],[87]. However, autonomous systems have to deal with abnormal behavior of a system as well. Thus, it is crucial to have a strong formalism and an environment that allows coherent integration of symbolic and numeric informations in a valid representation process to deal with our complex dynamic world [111].

This chapter presents a coherent methodology to integrate planning, operation, and diagnosis as developed in previous chapters. The chapter is organized as follows: First it briefly reviews each functional aspect of a high autonomy system. Then it describes a systematic methodology to integrate those aspects in a hierarchical fashion. This is then followed by an automatic system generation methodology illustrated by means of the example of our robot-managed space-borne laboratory.

10.2. Model Base Development

Figure 10.1 presents a model-based autonomous system architecture to integrate decision, action, and prediction components. The architecture features a model base at the center of its planning, operation, diagnosis, and fault recovery strategies. In this way, it integrates AI symbolic models and control-theoretic dynamic models into a coherent system.

Approaches to design various autonomous component models for planning, operation, and diagnosis have previously been developed in their respective research fields so that there are many overlaps as well as inconsistencies in assumptions. In an integrated system, such components cannot be considered independently. For example, planning requires execution, and diagnosis is activated when anomalies are detected during execution.

Planning is defined as "Reasoning about how to achieve a given goal." It employs a suitable model to map a connecting sequence of states from an initial state to a goal state within a normal operation envelope. Once a plan is set up, it should be faithfully executed. "Execution with verification" maps from the input command and its expected normal responses to success/failure. As long as the execution is successful, it continues until the goal is achieved. However, if it fails, the diagnosis function will be activated. Diagnosis is defined as "discovering the cause of a failure." It maps back from one or several observed symptoms to one or several plausible anomalies that may have caused the observed symptom(s). Having identified the causes, the autonomous system should be able to recover, i.e., plan a path from the faulty state of the real system to a normal state on the original plan.

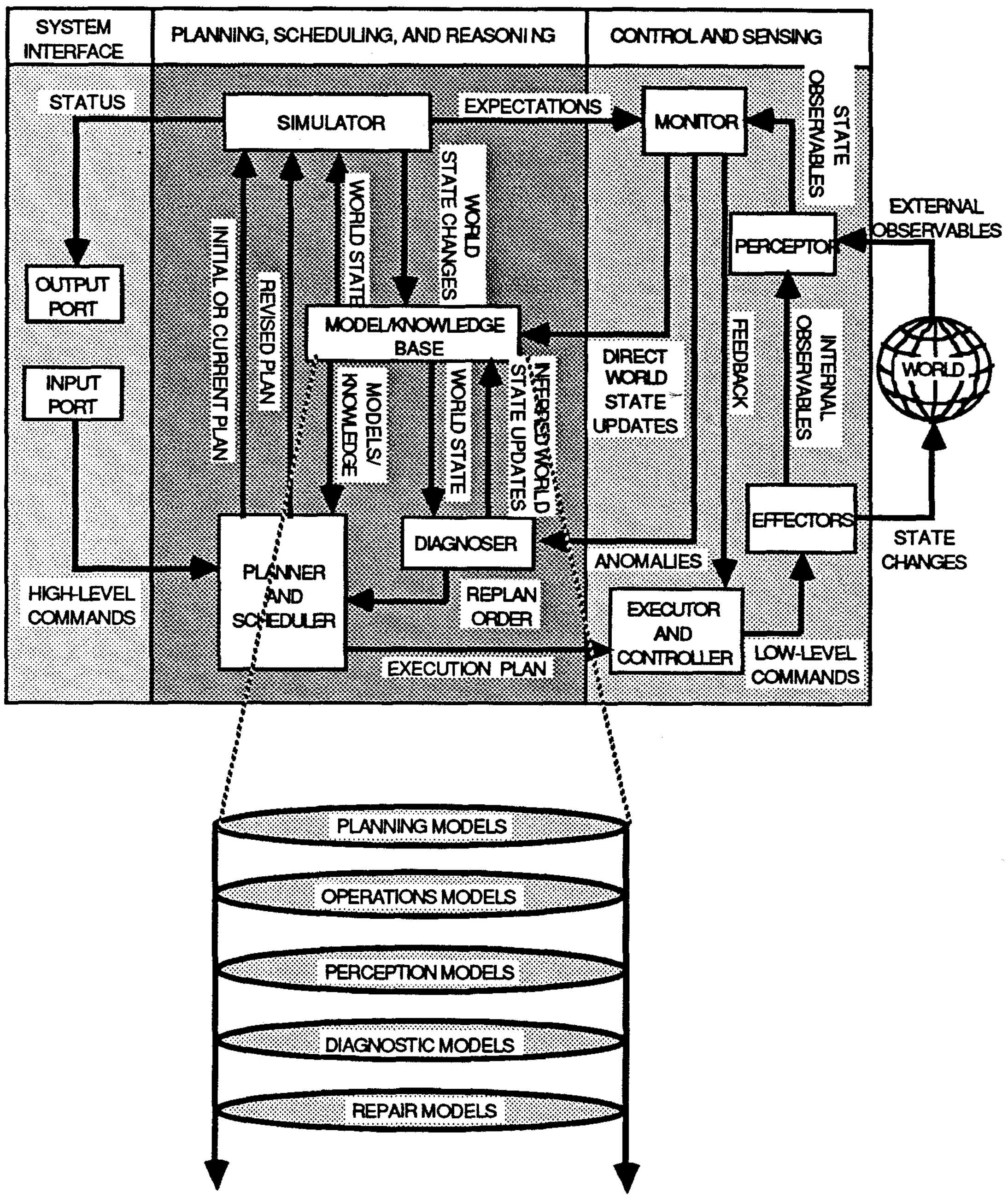


Figure 10.1. Autonomous System Architecture and its Model Base Class

10.2.1 Model-based Planning

There are two major approaches to task planning: one considers planning as searching and the other considers planning as a representation problem. The former deals with the initial planning problem where no prior experience is employed. In contrast, the latter, so-called case-based planning, views planning as remembering, i.e., retrieving and modifying existing plans for new problems [37].

In our model-based approach, planning is achieved by pruning a SES to select PESs from alternatives [17]. The PESs are in turn transformed into simulation model structures for execution. The non-experienced initial planning, which means the pruning of SES alternatives, can be achieved by using a rule-based approach. Every action (or state) node has several rules associated with system constraints, pre-conditions, and post-conditions. The resultant PES is saved with an index into an ENBASE for reuse. In contrast with non-experienced planning, the experienced planning is done by retrieving PESs from the ENBASE. The planner first retrieves a plan that might be used to achieve a given goal, or generates a new trial plan from partial plans if no existing plan is suitable. This candidate plan is then projected forward via simulation by attaching component models in a MBASE, where low level planning is embedded.

Once an execution model is synthesized, a lower level planner produces a goal table that is a list of 4-tuples: state, goal, command, and time-to-reach-goal. From these, a time optimal path from an initial state to a goal state is readily derived. Since discrete event models embody timing it is natural to base optimal sequencing on predicted execution time. The planner works by developing paths backward from the goal until the given initial states (possible starting states of the given system) are reached[14],[109].

10.2.2 Model-based Operation

The event-based control paradigm realizes intelligent control by employing a discrete eventistic form of control logic represented by the DEVS formalism [14],[102]. In this control paradigm, the controller expects to receive confirming sensor responses to its control commands within defined time windows determined by its model of the system under control. An essential advantage of the event-based operation is that the error messages it issues can bear important information for diagnostic purposes.

The operational model used by event-based operation has a state transition table that is abstracted from a more detailed model that represents both normal and abnormal behavior. The state transition table keeps a knowledge of states, commands, next states, outputs, and time windows. The window associated with a state is determined by bracketing the time-advance values of all transitions associated with the corresponding states in the more detailed model. This divergence arises due to variations in parameters and initial states considered to represent normal behavior.

The operator uses the goal-table from the planner and the state-table of its operational model to issue commands to the controlled device. When proper response signals are received the operator causes the model to advance to the next state corresponding to the one in which the device is supposed to be. The operator ceases interacting with the device as soon as any discrepancy, such as a *too-early* or *too-late* sensor response, occurs and calls on an associated fault diagnoser.

10.2.3. Model-based Diagnosis

A model-based diagnoser assumes that its model of the system is correct, and looks for discrepancies between the behavior of its model and the behavior of the real system. If such a discrepancy has been detected, it assumes that the system has undergone some change which is considered a fault. In our approach, the diagnoser will make use of a fault model to match the observed anomalous behavior of the system. Diagnosis in the model-based architecture is performed by local and global diagnosers, to find single or multiple faults using knowledge of structure and behavior.

By local diagnosis we mean the diagnostic description of a component model: the local diagnoser looks for a fault that might have occurred within the currently activated model unit. Once the controller has detected a sensor response discrepancy, the local diagnoser is activated. Data associated with the discrepancy, such as the state in which it occurred, and its timing, are also passed on to the local diagnoser. From such data, as well as information it can gather from auxiliary sensors, the local diagnoser tries to discover the fault that occurred.

If the local diagnosis fails, the global diagnoser is activated to analyze the enclosing coupled model . The global diagnostic model is a cause-effect table obtained by symbolic simulation to generate all fault-injected trajectories that reach states exhibiting the detected symptom. Faults injected in such trajectories represent candidate diagnoses [16].

10.3. Endomorphic System Concept

Endomorphism refers to the existence of a homomorphism from an object to a sub-object within it, the part (sub-object) then being a model of the whole [109]. As illustrated in Figure 10.2, in order to control an object, a high autonomy system needs a corresponding model of the object to determine the particular action to take. The internal model used by the system and its world base model are related by abstraction, i.e., some form of homomorphic (i.e., endomorphic) relation. The inference engine asks its internal model for the necessary information for interacting with the real world object. By "world base model" we mean the most comprehensive model of the world available to the system whether it exists as a single object or as a family of partial models in the model base.

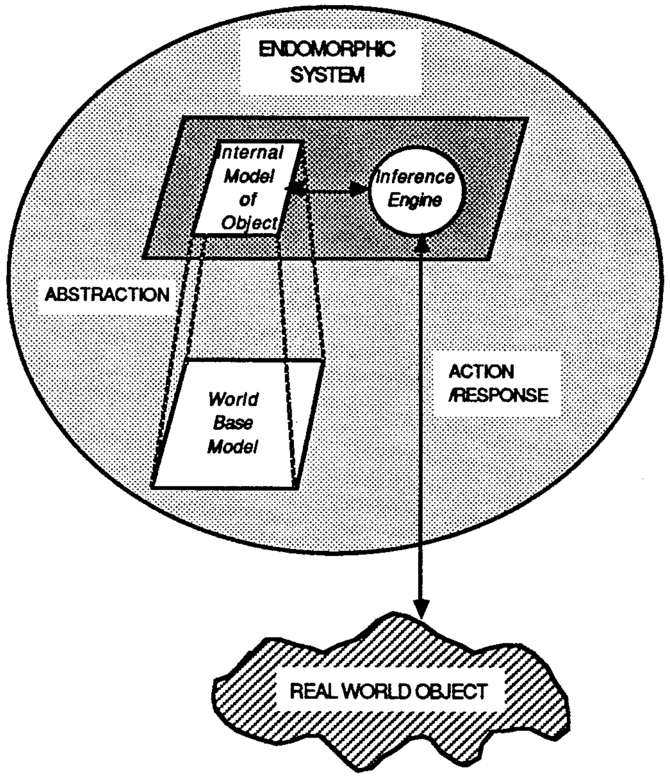


Figure 10.2. Endomorphic System Concept

10.4. Engine-based Design Methodologies

Typical expert systems comprise a domain-independent inference engine and a domain-dependent knowledge base. The inference engine examines the knowledge base and decides the order in which inferences are made. The engine-based modelling approach provides a clear separation between the domain-dependent model base and the domain-independent inference engine. It facilitates the automatic generation of a model base using endomorphisms. Figure 10.3 shows the engine-based modelling concept and examples of autonomous system components realized using the concept.

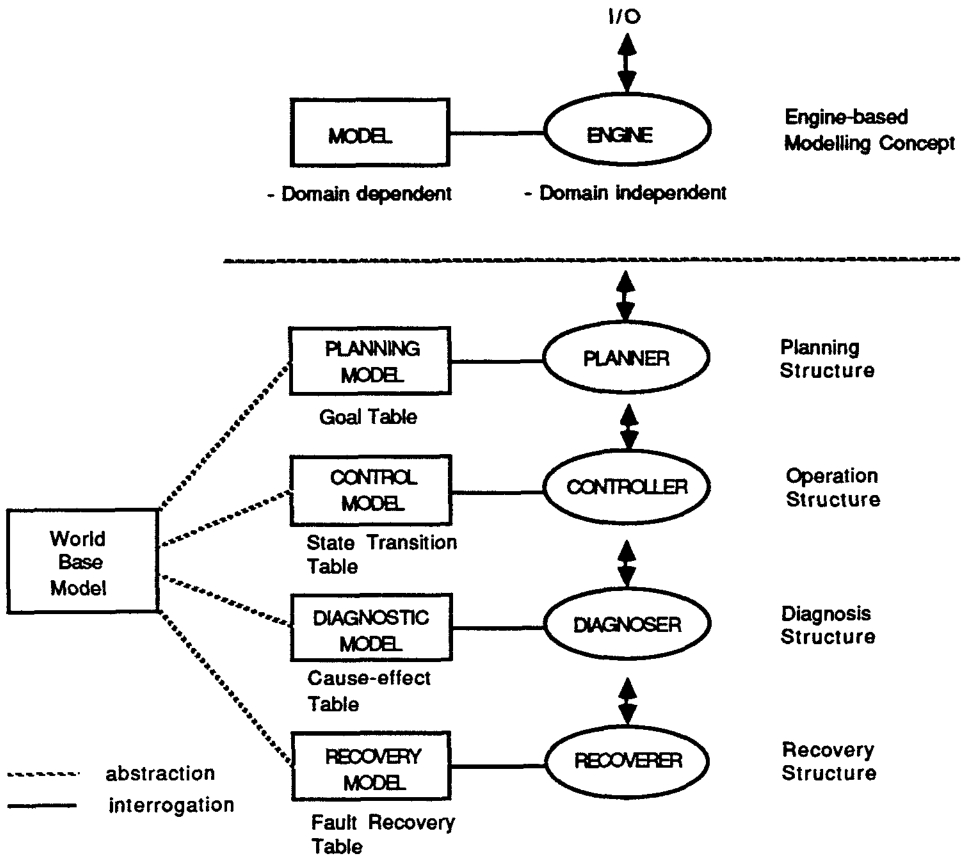


Figure 10.3. Engine-based Modelling Concept for Endomorphic System Design

10.5. Hierarchical Development of Intelligent Units

The autonomous system components described above have to be coupled within a unit in order to interact with each other. We use the term “intelligent unit” to denote the smallest unit that encapsulates all engine-based components as depicted in Figure 10.4.

To cope with complex problems, an autonomous system requires multiple intelligent units coupled in a hierarchical fashion. Models in the hierarchy must have valid abstraction relations to each other. Figure 10.4 illustrates an autonomous system development based on hierarchical abstraction and integration. Intelligent units at the leaf nodes of the execution structure employ internal models directly abstracted from the world base model. Units at higher levels employ internal models that are abstractions of coupled models composed of immediately inferior internal models.

The overall methodology for autonomous system generation in a model-based simulation environment is shown in Figure 8.3 of Chapter 8. The task formulation module receives an objective. It then retrieves an SES from the ENBASE and generates a plan structure by using the goal sub-SES of the SES (for initial planning) or partitioned PESs (for experience-based planning). A simulation structure is obtained by synthesizing the models in MBASE, where models can exist in advance or be generated automatically.

The initial environment for generating an autonomous system can be obtained using the layered development concept of DEVS-Scheme [109]. The first layer is the Lisp-based, object-oriented programming system that provides the foundation on which the system is built. The next layer is the SES/MB system where the behavioral and structural models can be built and saved in the MBASE and ENBASE, respectively. Models in MBASE are base models as well as internal models abstracted from base models.

Phase I: Plan generation

Once the basic environment is built, the next phase is the planning structure generation. When receiving a goal command, the task hierarchy is generated in a top-down fashion (task decomposition) as shown in Figure 10.5(a).

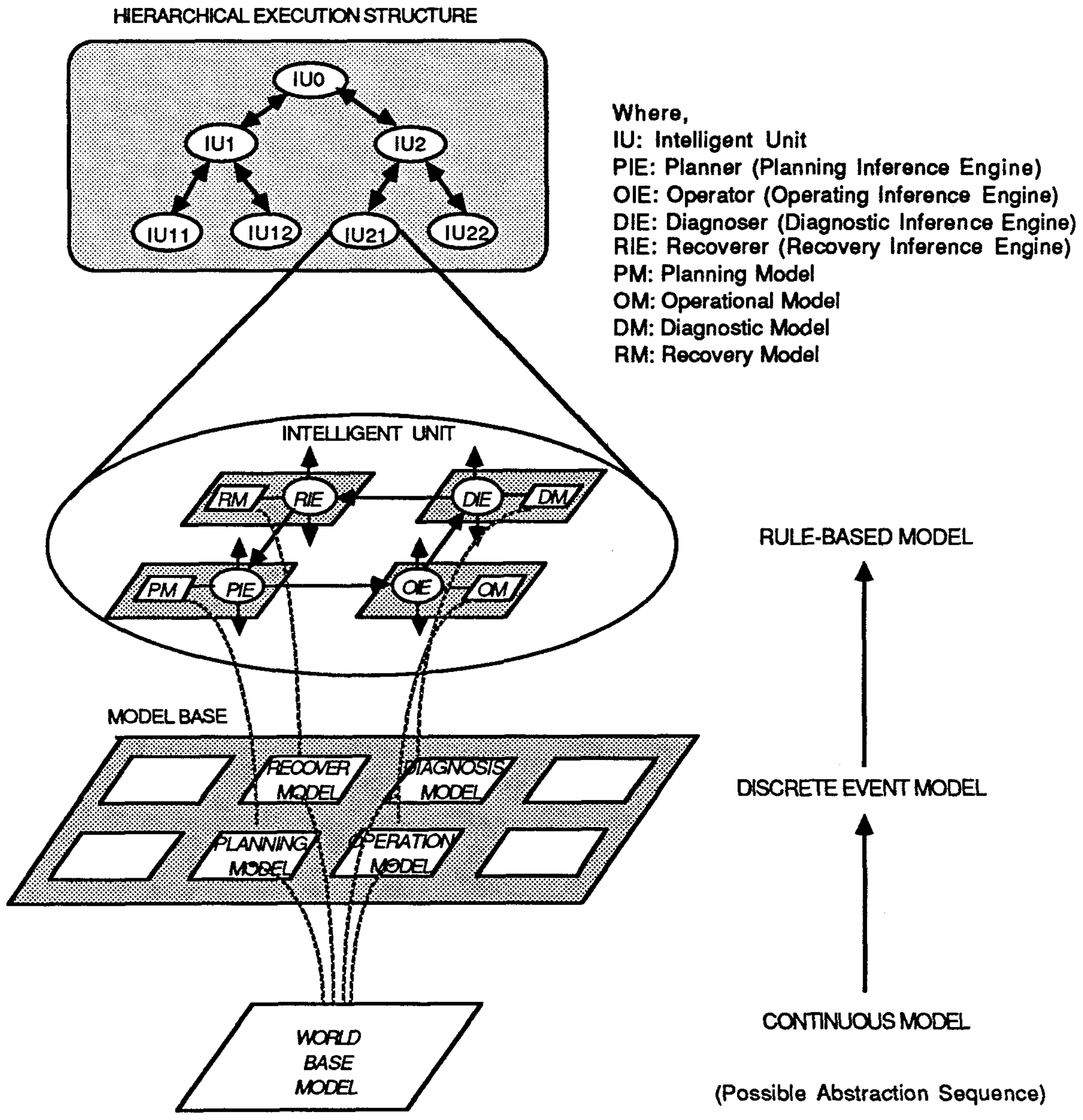


Figure 10.4. Hierarchical Abstraction and Integration of High Autonomy System

Phase II: Model construction

The next phase is the model base construction illustrated in Figure 10.5(b), where the necessary models can be retrieved from MBASE or automatically generated from the lower level models. This multi-layered hierarchical model generation and abstraction can be done in a bottom-up fashion. The resultant structure represents the domain dependent knowledge base structure.

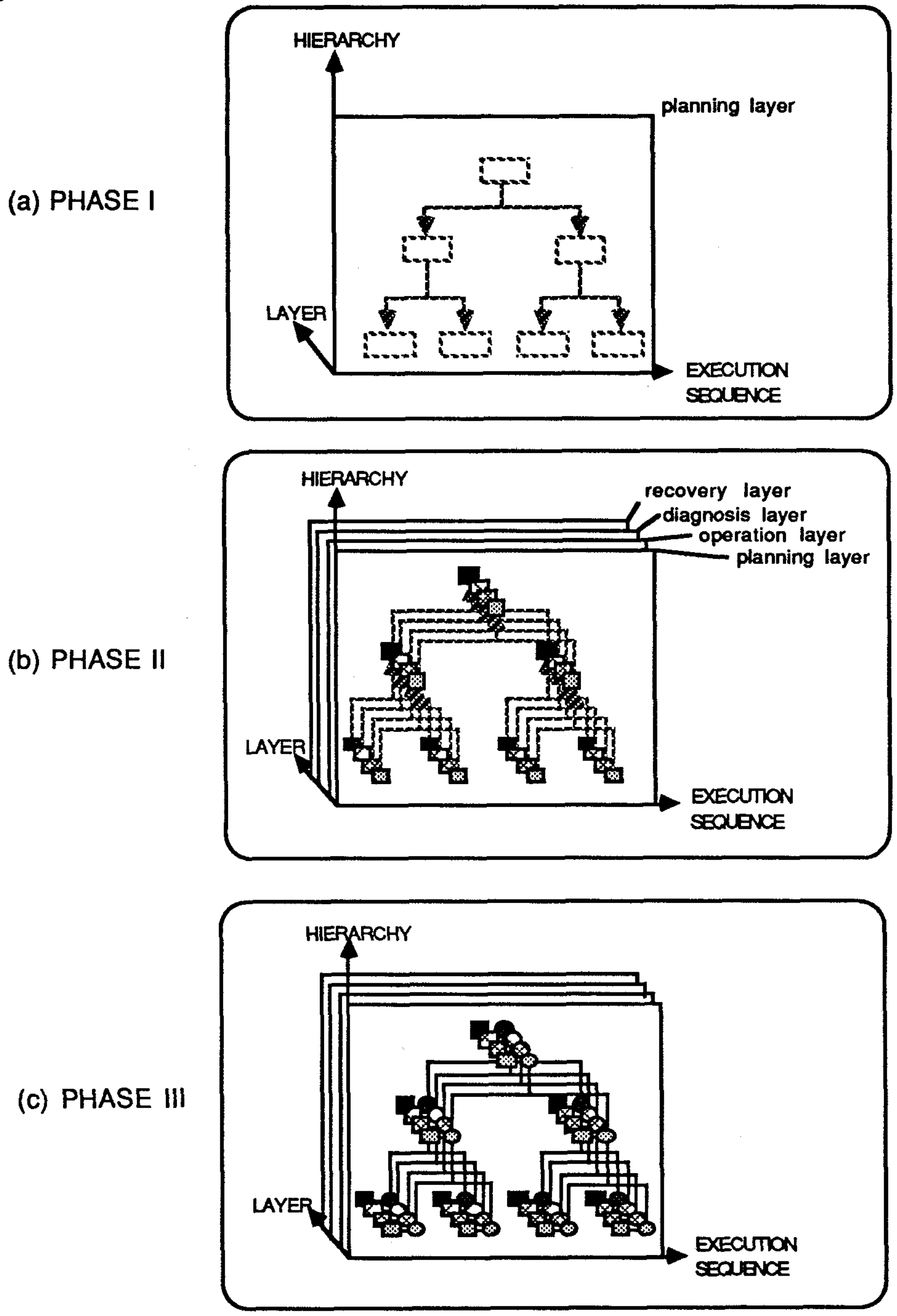


Figure 10.5. Autonomous System Generation Procedure

Phase III: Engine attachment/integration

By attaching domain independent engines such as a planner, an operator, a diagnoser, and a recoverer which are able to interrogate corresponding models, we have a multi-agent structure. Now by coupling those agents, we can obtain the autonomous system architecture shown in Figure 10.5(c).

10.6. Example: Robot-managed Fluid Handling Laboratory

As a concrete illustration, let us set up a mixing experiment (i.e., mix x amount of liquid-A with y amount of liquid-B) in the space-borne laboratory environment. To perform such a task, a robot must identify a syringe required to sample a liquid-A, then bring that syringe to the identified storage in which liquid-A is stored, and perform the sampling from the storage, and so on.

By either pruning the SES (no prior experience) employing the rule-based approach or retrieving existing PESs (experienced) in the EXPERIMENT part of the SES (see Figure 9), we can obtain a simulatable execution structure as depicted in Figure 10.6, where each MPU block shows its model name and its goal, respectively.

The execution is achieved by employing event-based control logic where it compares the expected time from the operational model with the observed time from the sensory response of the real system to be controlled. In this mixing example, after several successful executions, i.e. MOVE, PLACE, MOVE, and PLUG, suppose the actual filling time during the fill-it operation in SYRINGE LMPU is 12 seconds, but the operational model indicates 8 to 10 (minimum time is 8 seconds and window time is 2 seconds) as its normal timing, then we are in a fault situation and the detected fault timing is *too-late*.

Once the controller detects the fault, then the local diagnoser is activated, where it can use two types of sensory information: indicator (expected sensor, timing, phase) and vision (liquid-level, needle-angle, tube-constriction). If the sensory feedback information indicates: sensor = *full-sensor*, timing = *too-late*, phase = *filling*, angle = *normal* (i.e. 89 < angle < 91), constriction = *normal* (i.e. constriction < 10), and level < *100* (where 100 is the full level value for the small size syringe), then the local diagnosis in the SYRINGE unit would be "*Syringe is leaky*".

Note that our desired sub-plan of the SAMPLING unit is to sample a small amount of liquid-A from the storage-A. However, suppose a robot picks up a large size syringe instead of a small size syringe for some reason such as "arm motor abnormal" or "mobile motor abnormal" or "by obstacle", etc. If such an abnormal condition was not out of the detectable range within its action (model) unit, then such an abnormality would be propagated until the filling action where timing should be *too-late* because the actual syringe is the large size contrary to the intention. In this case, the sensory information would look as follows: sensor = *full-sensor*, timing = *too-late*, phase = *filling*, angle = *normal*, constriction = *normal*, and level ≥ *100*. Therefore the local diagnoser for the SYRINGE unit would not be able to conclude a fault. It would activate the global diagnoser in the SAMPLING unit. The global diagnoser uses the global diagnostic table that is compiled from the symbolic simulation of every possible fault case. It would conclude that the detected fault is caused by (1) position is changed during moving action to syringe desk so that robot position is in bad angle to pickup the large syringe during picking-up action or (2) position is changed during moving action to the liquid-A storage so that robot position is in bad angle to plug it in during plugging action, or (3) combination of both cases (see Figure 9.10). Among those possibilities, suppose case (1) has a higher priority (the priority may depend on the system requirements, objectives, etc.) than the other possibilities, then the recoverer would set up the new goal, for example, "remove obstacle," or "check power line," and then continue the remaining plan by picking up the small syringe.

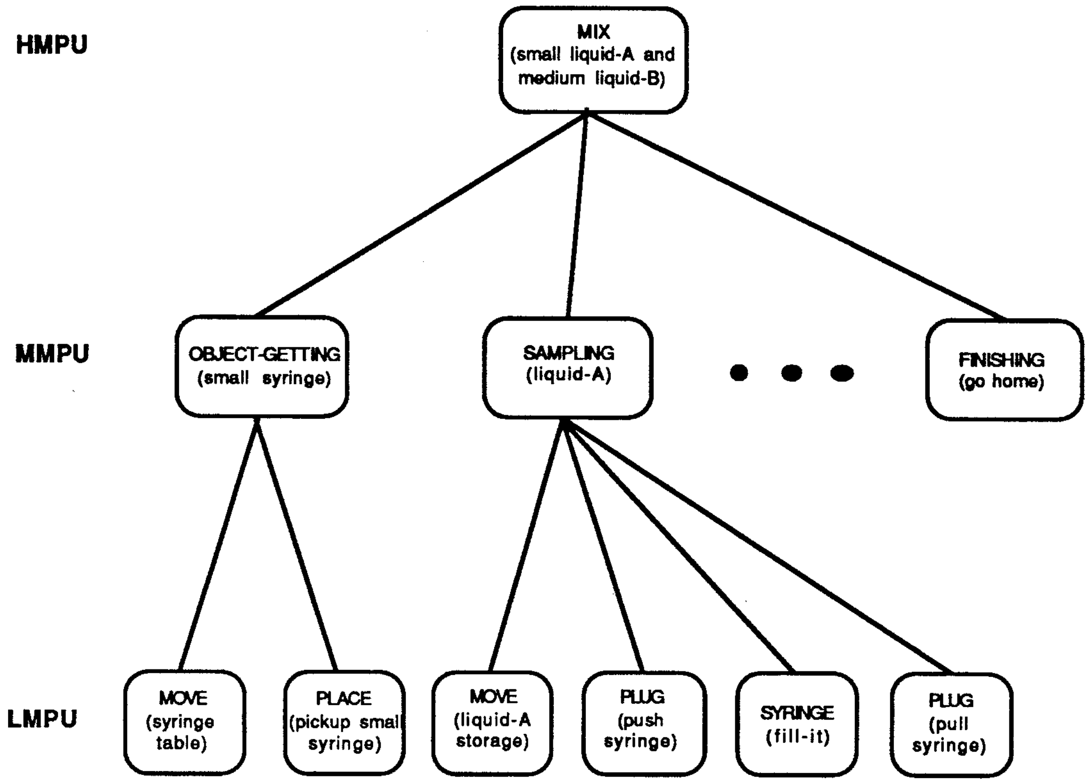


Figure 10.6. Mixing Example of Hierarchical Execution Structure

10.7. Conclusions

The main characteristics of the developed model-based architecture are as follows:

1. The time-based formalism (DEVS) provides coherent integration between symbolic and numeric models.
2. The SES/MB environment provides a hierarchical modularity, reusability, and testability.
3. Model-based deep reasoning supports a powerful diagnostic capability.
4. The endomorphism concept supports intelligent unit design and a consistent model base.
5. The engine-based design builds a domain-independent architecture instantiated with compiled models for application under real time constraints.
6. The intelligent unit encapsulates various autonomous components such as planning, operation, diagnosis, and recovery coherently to cope with complex problems.

7. Model-based planning supports the reusability of experienced plan structures.
8. The event-based control logic guarantees robustness, reduced sensor complexity, and increased diagnostic capability.

The framework has been implemented in the space-borne laboratory system as a proof of the concept.

CHAPTER 11
CONCLUSIONS

Building on the SES/MB framework as implemented in DEVS-Scheme, we have extended the ability of our knowledge/model base tools to support model-based design of high autonomy systems through the ability to build coherently a multi-agent, multi-function, multi-abstraction, multi-level architecture. We have developed a coherent methodology for constructing high autonomy systems by integrating high level symbolic models with low level dynamic models. The utility of the model-based approach for autonomous system design is demonstrated by means of models of a robot-managed fluid handling laboratory proposed for International Space Station Freedom.

The main advantages of the developed architecture in this dissertation are as follows:

A. The time-based formalism (DEVS) provides coherent integration between symbolic and numeric models.

B. The SES/MB environment provides hierarchical modularity, reusability, and testability.

C. Model-based deep reasoning supports a powerful diagnostic capability.

D. The endomorphism concept supports intelligent unit design and a consistent model base.

E. The engine-based design builds a domain-independent architecture instantiated with compiled models for application under real time constraints.

F. The intelligent unit encapsulates various autonomous components such as planning, operation, diagnosis, and recovery coherently to cope with complex problems.

G. Model-based planning supports the reusability of experienced plan structures.

H. The event-based control logic guarantees robustness, reduces sensor complexity, and increases diagnostic capability.

11.1. General Significance

Applications of autonomous systems such as flexible manufacturing systems, autonomous land and space vehicles, manned and unmanned planetary expeditions, and telerobotics are becoming an important factor in the technology of the future. Implementing systems autonomy is ultimately a systems engineering challenge: that of reducing the need for human intervention and supervision of systems. Space applications, where systems are to be deployed in remote hazardous environments, are a source of great scientific and industrial potential. They not only require high levels of autonomy to work but also have none of the job displacement side effects of robotics and automation on earth.

This research showed to develop architectural principles for design of autonomous systems whose behavior is based on models that support the various tasks that need to be performed. We proposed a model-based, hierarchical architecture aimed at reducing the computational demands required to integrate high level symbolic models with low level dynamic models. We sought to reduce on-line computation by off-line pre-compilation to produce simplified models individually matched to the tasks needing to be performed. Event-based control provided the basic methodology underlying model development and abstraction. Such models are to be attached to generic engines that can interpret them efficiently with short processing times.

The constructed working simulation version of an autonomous robot-managed laboratory demonstrates the use of multiple model families for experiment planning and execution including diagnosis. Tools to support development and integration of such model families are also developed. The developed model-based architecture is elaborated by incorporating time-based simulation and causal propagation model families supporting diagnosis, repair, and replanning. This involves tools to automatically extract such models from more detailed dynamic models and structural knowledge.

The utility of the proposed high autonomy system is demonstrated with models of a robot-managed fluid handling laboratory for International Space Station Freedom to be used for research in life sciences, microgravity sciences, and space medicine. NASA engineers will be able to base designs of intelligent controllers for such systems on our architectures developed in this dissertation. They will be able to employ our tools and simulation environment to verify such designs prior to implementation.

11.2. Contributions and Results

1. An extended discrete event specification formalism, symbolic DEVS, has been defined that extends the time base from the reals to the linear polynomials over the reals. This permits next event times to be expressed symbolically, although precise knowledge of any event time can be readily incorporated by setting its symbol to a real value. Therefore, symbolic DEVS is truly an extension of conventional DEVS and reduces to the latter when all times are real valued. A network of symbolic DEVS components is non-deterministic since the minimum of a set of next event times is not unique as in a conventional DEVS of the same kind. The extended formalism offers a convenient means to conduct multiple, simultaneous explorations of model behaviors.

2. To efficiently manage the symbolic DEVS, we have developed a linear polynomial constraint management algorithm for inferencing the constraint sets since symbolic times are linear polynomial expressions. The inference mechanism for handling linear equality/inequality constraints called consistency checking algorithm, based on the feasibility checking algorithm borrowed from linear programming, has been successfully developed and tested. In contrast with other reasoning approaches [83],[84],[88], our algorithm can efficiently and coherently integrate various reasoning approaches represented by linear polynomial forms with $>, <, =, \geq$, and $\leq$ relationships.

3. We have shown how symbolic DEVS can perform the causal propagation required for diagnosis of fault-prone systems modelled with multicomponent DEVS. By allowing the times to be expressed symbolically, we generate a family of trajectories that represent the possible sequences of events resulting in a particular breakdown. Such a family can be employed to trace a detected anomaly back to the components that might have caused it. The symbolic extension allows for both the situations where timings are unknown, or are known, but can vary. This approach to model-based diagnosis supersedes others [25], [44],[71] in that it intrinsically represents timing effects. Besides being applicable to fault diagnosis, symbolic DEVS may have a variety of applications where there is a need for efficient generation of a family of trajectories characterizing all scenarios of a given model.

4. We have shown how suitably operating on the structure of DEVS models provides a basis for the design of event-based logic control. In this control paradigm, the controller expects to receive confirming sensor responses to its control commands within predefined time windows determined by its DEVS model of the system under control. A space-adapted mixing control process was advantageously represented as a family of discrete event models by employing techniques based on the DEVS formalism. Several fluid handling models have been successfully tested in the DEVS-Scheme environment. An essential advantage of DEVS-based control is that the error messages it issues can bear important information for diagnostic purposes. Since the DEVS formalism is at the heart of event-based control system design, such controllers can be readily checked by computer simulation prior to implementation. Thus the DEVS formalism plays the same role with respect to event-based control that differential and difference equation formalisms play to conventional control. In contrast to other advanced control approaches [26],[38],[47], [54],[87], the DEVS methodology does not aim to provide improved graphical sequential logic specification languages that are concerned mainly with normal operation. While an

improved language such as GRAFCET [47],[54] facilitates maintenance and trouble shooting, the DEVS approach goes one step further: it is concerned with an integrated approach in which models can be developed to support all control aspects: planning, operation, diagnosis, and repair [14].

5. The SES/MB environment with the DEVS formalism provides various facilities to solve planning problems such as abstraction-related hierarchy, reusability, time constraints, replanning, and other extended planning issues [76],[88],[92]. The developed model-based planning system based on the SES/MB environment provides an extended planning paradigm, in which it can support not only the initial planning problem (searching problem) but also dynamic memory retrieval problems (representation problems). Moreover, it allows dynamic interaction with a simulation (execution) through operation and diagnosis. Building on the basis of the SES/MB framework, we have extended the ability of our knowledge/model base tools to support hierarchical model-based planning system design through the ability to generate families of planning alternatives, reusability of models, and model base coherence and evolvability.

6. The hierarchical model-based diagnosis system has been developed using simulation techniques. The local diagnoser can diagnose faults whose effects occur within the same component as the detected discrepancy. Faults whose effects propagate to other components are not locally diagnosable. To diagnose such faults we can use the global diagnostic engine and its associated model class. To do this, we adapted a symbolic simulation technique from linear programming to support automatic generation of all trajectories that are consistent with a detected operation anomaly. In contrast to Reiter's theory, we represent faults occurring in time, and analyze how systems respond to fault occurrences. This research has made several contributions: First, this approach to model-based diagnosis supersedes other approaches in that it intrinsically represents timing

effects. Second, the system is able to diagnose failures related to multiple faults. Third, it can discover all fault models as well as injected fault types that are consistent with the observed faulty behavior. Fourth, a clear separation is drawn between diagnostic engines and diagnostic models, resulting in a domain (and inference procedure) independent diagnostic procedure. Fifth, a separation of local and global diagnosers ensures a proper mapping onto the hierarchical nature of our intelligent system design, resulting in efficient diagnostic procedures. Sixth, the system is developed under the SES/MB modeling and simulation environment, resulting in a direct cooperation with other intelligent units such as the task planner, the event-based controller, and the repairer (replanner).

11.3. Future Research

Research needs to be continued to extend the developed methodology and environment to provide more powerful support for high autonomy system design. The following are suggested future research topics:

1. Extend the symbolic DEVS simulation capability such that it allows simulation of multi-level coupled-models. The abstract simulator implementation in DEVS-Scheme has been expanded so far to execute a symbolic simulation for single-level coupled-models with symbolic atomic-model components. The extended environment can serve as a deep reasoning basis for a variety of applications where there is a need of efficient generation of a family of trajectories characterizing all scenarios of a given model structure.

2. Develop error recovery strategies to enable to recover from a faulty state to a normal state. The functional aspects of high autonomy systems have been so far developed and implemented only for planning, operation, and diagnosis. However, the error recovery function will be needed to increase the levels of autonomy.

3. Provide a variable structure modelling and simulation environment to support a dynamic structuring capability. This facility will permit the creation, destruction, and modification of components during simulation runs. For example, modelling of high autonomy systems may require components within a model to be assembled and disassembled. Exchange of components within different sub-trees of a model may be involved [109].

4. Enhance the design capability by integrating real-time DEVS [96] for applicable interactions with sensors and actuators, the morphism preserving abstraction process [56] for the automated model generation, and the continuous system interface [89] for obtaining discrete eventistic characteristics of a system.

REFERENCES

[1] Albus, J.S., "A Theory of Intelligent Systems," Proc. IEEE Conf. on Intelligent Control, September, Philadelphia, PA, 1990.

[2] Albus, J.S., "Hierarchical Interaction Between Sensory Processing and World Modeling in Intelligent Systems," Proc. IEEE Conf. on Intelligent Control, September, Philadelphia, PA, 1990.

[3] Albus, J.S., "The Role of World Modeling and Value Judgment in Perception," Proc. IEEE Conf. on Intelligent Control, September, Philadelphia, PA, 1990.

[4] Albus, J.S., H.G. McCain, and R. Lumia, *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*, NBS Technical Note 1235, Robot Systems Division, Center for Manufacturing Engineering, National Technical Information Service, Gaithersburg, MD, 1987.

[5] Allen, J.F., "Toward a General Theory of Action and Time," *Artificial Intelligence*, Vol. 23, pp. 123-134, 1984.

[6] Antsaklis, P.J., K.M. Passino and S.J. Wang, "Towards Intelligent Autonomous control Systems: Architecture and Fundamental Issues," *J. Intelligent and robotics systems,* Vol.1, No.4, pp. 315-342, 1989.

[7] Buchanan, H.G. and E.H. Shortliffe, *Rule-Based Expert Systems*, Addison-Wesley, CA, 1984.

[8] Brooks, R.A., "AA Robust Layered Control System for a Mobile Robot," *IEEE J. of Robotics and Automation*, Vol. RA2, pp. 14-23, 1986.

[9] Cassandras, C.G. and S.G. Strickland, "Sample Path Properties of Timed Discrete Event Systems," Proc. of the IEEE, Vol. 77, No. 1, pp. 59-71, Jan., 1989.

[10] Cellier, F.E., "Combined continuous/Discrete simulation Application, Techniques and Tools," proc. of the 1986 winter simulation conference Washington D.C. 1986.

[11] Cellier, F.E., *Continuous System Modeling*, Springer-Verlag, NY, 1991.

[12] Chandrasekaran, B., A roundtable discussion: Present and Future directions: trends in Artificial Intelligence, OE Reports, SPIE, September, 1989.

[13] Chi, S.D., J.W. Na, and J.W. Kim, "Planning Methodologies: Survey and Model-based Approach," Term Paper, ECE576 Class, Dept. of Electrical and Computer Engr., University of Arizona, Spring, 1990.

[14] Chi, S.D. and B.P. Zeigler, "DEVS-based intelligent Control of Space Adapted Fluid Mixing," Proceeding of 5th conf. on Artificial Intelligence for Space Applications, May, 1990.

[15] Chi, S.D. and B.P. Zeigler, "Linear Polynomial Constraints Checking Algorithm for Symbolic Reasoning," in preparation for *Artificial Intelligence.*

[16] Chi, S.D. and B.P. Zeigler, "Model-based Hierarchical Diagnosis for High Autonomy System," in preparation.

[17] Chi, S.D., B. P. Zeigler, and F. E. Cellier, "Model-based Task Planning System for a Space Laboratory Environment," *SPIE Conference on Cooperative Intelligent Robotics in Space*, Boston, Nov., 1990.

[18] Chien, G., "Dynamic System Modeling and Simulation in Product Design," Master Thesis, Illinois Inst. of Tech., 1989.

[19] Cohen, P.R. and E.A. Feigenbaum, *The Handbook of Artificial Intelligence*, Vol. 3, Los Altos, CA: W. Kaufmann, 1982.

[20] Combacau, M. and M. Courvoiser (1990), "A Hierarchical and Modular Structure for FMS Control and Modelling," *Proc. AI, Simulation, and Planning in High Autonomy Systems*, IEEE Press, Tucson, March 26-27, pp. 204-211, 1990.

[21] D'Azzo, J. and C.H. Houpis, *Linear Control System Analysis and Design: Conventional and Modern* 3rd Ed., McGraw Hill, York, 1988.

[22] Davis, E., "Constraint Propagation with Interval Labels", *Artificial Intelligence*, Vol. 32, pp. 281-331, 1987.

[23] Davis, R., "Diagnostic reasoning based on structure and behabior", *Artificial Intelligence*, Vol. 24, pp. 347-410, 1984.

[24] DeKleer, Y., "Qualitative Physics, A Personal View," In: *Readings in Qualitative Physics* (eds. D. Weld and Y. dekleer), Morgan Kaufman, Palo Alto, 1989.

[25] DeKleer. Y. and B.C. Williams, "Diagnosing multiple faults", *Artificial Intelligence*, Vol. 32, pp. 97-130, 1987.

[26] Fanard, A.G., M.C. Lobelle, and E.B. Mulemangabo, "G++, A Graphical Language Intended to Help the Development of Industrial Process Control Applications," 2nd Int. Conf. on Software Engineering for Real Time Systems, pp. 85-89, Sept., 1989.

[27] Fischler, M.A. and O. Firshein, *Intelligence, The Eye, The Brain, and The Computer*, Addison-Wesley Pub. Co., Reading, MA, 1987.

[28] Fishwick, P.A., "Fuzzy Simulation: Specifying and Identifying Qualitative Models," Special Issue on Modelling and Simulation for High Autonomy Systems, *Int. J. Gen. Sys.* (to be appear).

[29] Fishwick, P.A., "A Taxonomy for Process Abstraction in Simulation Modelling," *IEEE Int. Conf. Sys. Man & Cyb.*, Vol. 1, pp. 144-151, 1987.

[30] Fishwick, P.A., "Abstraction Level Traversal in Hierarchical Modelling," In: *Modelling and Simulation Methodology: Knowledge Systems Paradigms* (eds.: M.S. Elzas, T.I. Oren, B.P. Zeigler), North Holland Pub. Co., Amsterdam, pp. 393-430, 1989.

[31] Garzia, R.F., M.R. Garzia, and B.P. Zeigler, "Discrete Event Simulation," *IEEE Spectrum*, December, pp. 32-36, 1986.

[32] Genesereth M.R. and Nilsson, N.J., *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Pub. Los Altos, CA, 1987.

[33] Gensereth, M.R., "The Use of Design Descriptions in Automated Diagnosis," *Artificial Intelligence*, Vol. 23, pp. 411-436, 1984.

[34] Gevarter, W.B., *Intelligent Machines*, Prentice-Hall, New Jersey, 1985.

[35] Glynn, P.W., "A GSMP Formalism for Discrete Event Systems," Proc. of the IEEE, Vol. 77, No. 1, pp. 14-23, Jan., 1989.

[36] Greiner, R., B.A. Smith and R.W. Wilkerson, "A correction to the algorithm in Reiter's theory of diagnosis", *Artificial Intelligence*, Vol. 41, pp. 79-88, 1989.

[37] Hammond, K.J.,*Case-Based Planning*, Academic Press, 1989.

[38] Harel, D., "On Visual Formalisms," Comm. of ACM, Vol. 31, No. 5, pp. 514-530, May, 1988.

[39] Hamscher W. and R. Patil, "Tutorial on Model-based Diagnosis," Proc. of 11th IJCAI, Detroit, 1989.

[40] Hawker, J.S. and R.N. Nagel, "World Models in Intelligent control Systems," IEEE Int. Symposium on Intelligent control, pp.482-488, 1987.

[41] Ho, Y. "Editors Introduction," *Special Issue on Dynamics of Discrete Event Systems, Proceeedings of IEEE*, Vol. 77, No. 1, 1989.

[42] Hu, J. and J.W. Rozenblit, "Knowledge Acquisition Based on Representation for Design Model Development," In: *Knowledge-based Simulation methodology and Applications* (eds.P.A. Fishwick and R.D.Modkeski), Springer Verlag, Berlin, 1989.

[43] Hurst, W.J. and Mortimer, J.W., *Laboratory Robotics : A Guide to Planning Programming and Applications,* VCH Publishers, 1987.

[44] Iwasaki, Y., "An Integrated Scheme for Using First-Principle Physical Knowledge for Knowledge-based Simulation," *Proceedings of Fourth AAAI Workshop on AI and Simulation*, pp. 57-59, 1989.

[45] Kanade, T., A roundtable discussion: Present and Future directions: trends in Artificial Intelligence, OE Reports, SPIE, September, 1989.

[46] Kapila, A.K., *Asymptotic Treatment of Chemically Reacting Systems*, Pitman Publishing Inc., Marshfield, Massachusetts, 1983.

[47] Keller, W.L., "GRAFCET, A Functional Chart for Sequencial Processes," Proc. on 14th Annual Int. Programmable Controllers, pp. 71-96, Detroit, Michigan, April, 1985.

[48] Kim, T.G. and B.P. Zeigler, "ESP-Scheme: A realization of system Entity Structure in a LISP Environment," Proc. in 1989 SCS Eastern Multiconference, Tempa, Florida, March, 1989.

[49] Kim, T.G. and B.P. Zeigler, "Hierarchical Scheduling in an Intelligent Environmental Control System," *Proc. 2nd Int. Conf. on Industrial and Engineering Applications of AI and Expert Systems*, 1988.

[50] Kim, T.G., "A knowledge-based Environment for Hierarchical Modelling and Simulation," Ph.D Dissertation, Univ. of Arizona, 1988.

[51] Klir, G., *Architecture of Systems Problem Solving*, Plenum Press, NY, 1985.

[52] Kuipers, B.J., "Qualitative Reasoning with Causal Models in Diagnosis of Complex Systems," In: *Artificial Intelligence, Simulation and Modelling* (eds. L.A. Widman, K.A. Loparo, and N. Nielsen), J. Wiley, NY, pp. 257-274, 1989.

[53] Kuipers, B.J., "Qualitative Simulation," *Artificial Intelligence*, pp. 289-338, 1986.

[54] Lioyd, M., "GRAFCET - Graphical Function Chart Programming," Proc. of the Conf. on Programmable Controllers '85, pp. 51-56, London, Olympia, July, 1985.

[55] Luh, C.J., "Abstraction Morphisms for High Autonomy Systems," Ph.D. Dissertation, Univ. of Arizona, 1991.

[56] Luh, C.J. and B.P. Zeigler, "Abstraction Morphisms for Muitifacetted Methodolody: Application to Model-Based Autonomous Systems," (submitted to *IEEE. Trans. Sys. Man. & Cyber.*), 1990.

[57] Mesarovic, M.D. and Y. Takahara, *General Systems Theory: Mathematical Foundations*, Academic Press, NY, 1975.

[58] Meystel, A., "Intelligent Control: A Sketch of the Theory," *J. Intelligent and Robotic Systems*, vol.2, No.2&3, pp.97-107, 1989.

[59] Meystel, A., "Intelligent Control: Highlights and Shadows," *Proc. IEEE on Intelligent Control*, Philadelphia, Jan., 1987.

[60] Meystel, A., "Planning/Control Architecture for Master Dependent Autonomous Systems with Nonhomogeneous Knowledge Representation," *Proc. IEEE on Intelligent Control,* Philadelphia, Jan., 1987.

[61] Nance, R.E., "A Conical Methodology: A Framework for Simulation Model Development," *Proc. Conf. on Methodology and Validation*, SCS Pubs. San Diego, pp. 38-43, 1987.

[62] Narain, S. "An Approach to Reasoning about Hybrid Systems," Special Issue on Modelling and Simulation for High Autonomy Systems, *Int. J. Gen. Sys.* (to appear).

[63] NASA, *The Space Station Program*, NASA Pub., 1985.

[64] Newell, A., "Putting it All together," In: *Complex Information Processing: The Impact of Herbert A. Simon* (eds: D. Klahr, K. Kotovosky), Lawerence Erlbaum, Hillsdale, NJ, 1988.

[65] Nilsson, N.J., *Principles of Artificial Intelligence*, Tioga Pub. Co., Palo Alto, CA, 1981.

[66] Ogata, K., *Modern Control Engineering*, Prentice-Hall, 1970.

[67] Orden, A., "On the Solution of Linear Equation/inequality Systems", *Mathematical Programming*, Vol.1, pp. 137-152, 1971.

[68] Oren, T.I., "Taxonomy of simulation model processing," in *Encyclopedia of Systems and Control*, M. Signh, Ed. New York, NY : Pergamon Press.

[69] Praehofer, H., "System Theoretic Formalisms for Combined Discrete-Continuous System Simulation," Special Issue on Modelling and Simulation for High Autonomy Systems, *Int. J. Gen. Sys.* (to appear).

[70] Rajagopalan, R., "The role of Qualitative Reasoning in Simulation," In: *Artificial Intelligence in Simulation* (eds. G.C. Vansteenkiste, E.J.H. Kerckhoffs, & B.P. Zeigler), SCS Pub., San Diego, CA, 1986.

[71] Reiter, R., "Theory of diagnosis from first principles", *Artificial Intelligence*, Vol. 32, pp. 57-95, 1987.

[72] Rich, E., *Artificial Intelligence*, 1986.

[73] Riesbeck, C.K. and R.C. Schank, *Inside Case-Based Reasing,* Lawrence Erlbaum Associates, Inc., Hillside, New Jersey, 1989.

[74] Rozenblit, J.W. and B.P.Zeigler, "Design and Modelling Concepts," *Encyclopedia of robotics*, John Wiley, N.Y.,1987.

[75] Rozenblit, J.W., S. Sevinc and B.P. Zeigler, "Knowledge-Based Design of LANs Using System Entity Structure Concepts," *Proc. Winter Simulation Conf.*, Washington D.C., 1986.

[76] Sacerdoti, E.D., *A Structure for Plans and Behavior,* Elsevier North-Holland, Inc., 1977.

[77] Sanders, W.H., *Construction and Solution of Performability Models based on Stochastic Activity Networks*, Doctoral Diss., University of Michigan, Ann Arbor, 1988.

[78] Saridis, G.N., "Intelligent Robotic controls," *IEEE Trans. on Auto. control.*, AC-28, No.5,1983.

[79] Saridis, G.N., "Knowledge Implementation: Structure of Intelligent Control System," *Proc. IEEE on Intelligent Control*, Philadelphia, Jan., 1987.

[80] Sarjoughian, H.S., F.E. Cellier and B.P. Zeigler, "Hierarchical Controllers and Diagnostic Units for Semi-Autonomous Teleoperation of a Fluid Handling Laboratory," Proc. Int. Pheonix conf. on Computer and communication, Scottdale, March, 1990.

[81] Scarl, E., "Sensor Failure and Missing Data: Further Inducements for Reasoning with Models," (personal communication), 1989.

[82] Sevinc, S and B.P. Zeigler, "Entity Structure Based Design Methodology: A LAN Protocol Example," *IEEE Trans. on Software Engineering*, Vol. 14, No. 3, March, pp. 375-383, 1988.

[83] Simmons, R. "Integrating Multiple Representations for Incremental, Causal Simulation," *Proc. AI, Simulation and Planning in High Autonomy Systems*, pp. 88-95, Cocoa Beach, FL, April, 1991.

[84] Simmons, R. "Commonsense Arithmetic Reasoning," Proc. AAAI-86, pp. 118--124, Philadelphia, PA, August, 1986.

[85] Soucek, B. and A.D. Carlson, "Brain Window Language of fire-flies," *J. Theor. Biol.*, vol. 125, pp. 93-103, 1987.

[86] Steel, S., "Topics in Planning," in Lecture Notes in Artificial Intelligence, J. Siekmann eds., Springer-Verlag, 1987.

[87] Struger, O.J., "Programmable Controllers - Past and Future," Proc. of the Conf. on Programmable Controllers '85, pp. 1-6, London, Olympia, July, 1985.

[88] Vere, S.A., "Planning in Time : Windows and Durations for Activities and Goals," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. PAMI-5, No. 3, pp. 246-267, May, 1983.

[89] Wang, Q. and F.E. Cellier, "Time Windows : An approach to Automated Abstraction of Continuous-Time Models into Discrete-Event Models," *Int. J. of General Systems,* special issue on modeling and simulation of high autonomy systems, to appear, 1991.

[90] Waterman, D.A., *A Guide to Expert Systems*, Addison-Wesley Publishing Co., 1985.

[91] Widman, L.E., “Semi-Qualitative "Close-Enough" Dynamic Systems Models: An Alternative to Qualitative Simulation,” In: *Artificial Intelligence, Simulation and Modelling* (eds.: L.A. Widman, K.A. Loparo, and N. Nielsen), J. Wiley, NY, pp. 159-188.

[92] Wilkins, D.E., *Practical Planning*, Morgan Kaufmann Inc., 1988.

[93] Wymore, A.W., *A Mathematical Theory of Systems Engineering: The Elements*, Wiley, NY, 1967.

[94] Zadeh, L.A. and C.A. Desoer, *Linear System Theory, The State Space Approach*, McGraw Hill, NY, 1963.

[95] Zeigler, B.P. and G. Zhang, “Formalization of the System Entity Structure Knowledge Representation Scheme : Proofs of Correctness of Transformations,” In : *AI, Simulation, and Modelling*, L. Widman, D. Reidel (Eds.), 1987.

[96] Zeigler, B.P. and J. Kim, “Extending The DEVS-Scheme Knowledge-Based Simulation Environment For Real-Time Event-Based Control,” (submitted to *IEEE Trans. Aut. and Rob.*), 1991.

[97] Zeigler, B.P. and S. D. Chi, “Symbolic Discrete Event System Specification,” *Proc. on AI, Simulation and Planning in High Autonomy Systems*, pp. 130-141, IEEE Press, April, 1991.

[98] Zeigler, B.P. and S.D. Chi, “Hierarchical Systems Architecture for Artificial Intelligence,” *Proc. on 34th Int. Soc. System Sciences Conf.*, Portland, July, 1990.

[99] Zeigler, B.P. and S.D. Chi, “Model-Based Architectures for Autonomous Systems,” *Proc. 1990 IEEE Symp. on Intelligent Control*, Philadelphia, PA, pp. 27-32, 1990.

[100] Zeigler, B.P. “Systems Formulation of a Theory of Diagnosis from First Principles,” in preparation.

[101] Zeigler, B.P., C.J. Luh, and T.G. Kim, “Model Base Management for Multifacetted Systems,” Proc. AI, Simulation and Planning in High Autonomy Systems, pp. 25-35, IEEE Press, Tucson, March, 1990.

[102] Zeigler, B.P., “DEVS Representation of Dynamical Systems: Event-Based Intelligent Control,” IEEE proc. Vol.77, no.1, pp.72-80, Jan., 1989.

[103] Zeigler, B.P., F.E. Cellier and J.W. Rozenblit, “Design of a simulation Environment for laboratory management by Robot organizations,” *J. Intelligent and Robotic systems*, vol.1, pp. 299-309, 1988.

[104] Zeigler, B.P., “Hierarchical, modular discrete event modelling in an object oriented environment,” *Simulation*, Vol. 49, no.5, pp. 219-230, 1987.

[105] Zeigler, B.P., “High Autonomy systems: Concepts and Models,” *Proc. on AI, simulation and Planning in High autonomy Systems*, Tucson, march, 1990.

[106] Zeigler, B.P., "Implementation of Methodology Based Tools in the DEVS-Scheme Environment," In : *Modelling and Simulation Methodology : Knowledge System Paradigms,* M. S. Elzas, T. I. Oren, B. P. Zeigler (Eds.), North-Holland, Amsterdamm, 1988.

[107] Zeigler, B.P., "Knowledge Representation from Minsky to Newton and Beyond," Applied Artificial Intelligence, vol. 1, pp 87-107, Hemisphere Pub., Co., 1987.

[108] Zeigler, B.P., *Multifaceted Modelling and discrete Event simulation,* Academic press, 1984.

[109] Zeigler, B.P., *Object-Oriented simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic systems,* Academic Press, 1990.

[110] Zeigler, B.P., R. Doyle, E. Scarl, F. Lacklinger, S.D. Chi, and S. Feycock, "Model-Based Sensor Selection, Monitoring, and Diagnosis," *IEEE Expert,* 1992 (to appear).

[111] Zeigler, B.P., S.D. Chi, and F.E. Cellier, "Model-based Architecture for High Autonomy Systems," *Proc. European Robotics and Intelligent Systems Conf.*, Corfu, Greece, June, 1991.

[112] Zeigler, B.P., "System-theoretic representation of simulation models," *IIE Trans.*, pp. 19-34, Mar., 1984.

[113] Zeigler, B.P., *Systems Simulateable by the Digital Computer: Discrete Event Representable Models,* Tech. Rept. 199, Dept. Computer Sci., University of Michigan, Ann Arbor, MI, 1977.

[114] Zeigler, B.P., "Toward a Simulation Methodology For Variable Structure Modelling," in *Modelling and Simulation Methodology in the Artificial Intelligence Era,* M.S. Elzas, B. P. Zeigler, and T. I. Oren Eds., North-holland, 1986.

[115] Zeigler, B.P.,*Theory of Modelling and Simulation,* New York, NY : Wiley, 1976 (reissued by Krieger Pub. Co., Malabar, FL,1985).

[116] Zhong, H. and M.J. Wonham, "Hierarchical Coordination," *Proc. 5th EEE Symp. on Intelligent Control,* Philadelphia, PA, pp. 8-14, 1990.