GIBSS: A FRAMEWORK FOR THE MULTI-LEVEL SIMULATION OF MANUFACTURING SYSTEMS

by

Edward Christopher De Meter

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy
in
Industrial Engineering and Operations Research

APPROVED:

Michael P. Deisenroth, Ph.D, Chairman

Robert G. Leonard, Ph.D. H. Jo Anne Freeman, Ph.D., P.E.

Roberta S. Russell, Ph.D. Konstantinos P. Triantis, Ph.D.

June 1989

GIBSS: A FRAMEWORK FOR THE MULTI-LEVEL SIMULATION OF MANUFACTURING SYSTEMS

by

Edward Christopher De Meter

Industrial Engineering and Operations Research

(ABSTRACT)

A systems approach for manufacturing system design calls for the division of a system design into sub-designs, and their specification over multiple levels of detail. Through an iterative design and evaluation process, a system design progresses from an abstraction to an implemental specification. To facilitate the evaluation process, models of sub-designs must be applicable to modular assembly, even if the sub-designs are heterogeneously specified.

Computer simulation modeling is currently the most flexible method of manufacturing system analysis. When used in the multi-level design process, two forms of simulation models are encountered, uni-level and multi-level. A simulation model of a manufacturing system is considered uni-level if objects of equivalent type within the system are modeled at the same level of detail. On the other hand, a model is considered multi-level if objects of equivalent type are not modeled at the same level of detail. Unfortunately, current simulation frameworks do not integrate modular construction with the various discrete event and continuous simulation techniques needed to support multi-level modeling.

This dissertation describes **GIBSS** (Generalized Interaction Based Simulation Specification), a simulation framework which supports the modular construction of uni-level and multi-level simulation models. Under GIBSS, the mechanisms and attributes of a manufacturing system simulation are distributed among various classes of independent sub-models. These classes are **passive**, **internal interaction**, **external interaction**, and **master simulation**. GIBSS describes the mechanics of each of these classes, as well as their method of synchronization. Using GIBSS, sub-models are created, executed, and validated independently, and then brought together to execute in parallel or near parallel fashion. As a result, uni-level and multi-level system simulation models are assembled from multiple sub-models.

GIBSS eliminates a barrier to the rapid evaluation of manufacturing system designs. It facilitates the multi-level design process, and is the basis of a research effort, dedicated to the development of a new generation of computer-aided manufacturing system design environments.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Dr. Michael P. Deisenroth. Over the past five years, he has served as my major advisor and provided me with the opportunity to work and expand my technical education in many areas. More importantly, he has been a good friend, who has continuously demonstrated the value of hard work, resourcefulness and integrity. Without question, academia would improve drastically if more faculty were like him.

In addition, I would like to thank my wife, , and children, and , for their unlimited patience. I know it has been difficult over the past three years living with a part time husband and father. I would also like to thank my parents, , for their unending support, and 's parents, , for treating me like a son and helping out with the kids in my absence. In particular, I regret the loss of , whom I deeply respected.

Finally, I would like to thank , with whom I have enjoyed many a philosophical discussion over the past three years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.0 Introduction to Research Topic

A systems approach for manufacturing system design calls for the division of a system design into sub-designs, and their specification over multiple levels of detail. Through an iterative design and evaluation process, a system design progresses from an abstraction to an implemental specification. To facilitate the evaluation process, models of sub-designs must be applicable to modular assembly, even if the sub-designs are heterogeneously specified.

Computer simulation modeling is currently the most flexible method of manufacturing system analysis. When used in the multi-level design process, two forms of simulation models are encountered, uni-level and multi-level. A simulation model of a manufacturing system is considered uni-level if all objects of equivalent type within the system are modeled at the same level of detail. On the other hand, a model is considered multi-level if objects of equivalent type are not modeled at the same level of detail. Unfortunately, current simulation frameworks do not integrate modular construction with the various discrete event and continuous simulation techniques needed to support multi-level modeling. As a result, the use of simulation in the multi-level design process is impeded.

This dissertation presents a simulation framework which supports the modular construction of uni-level and multi-level simulation models, and in turn supports the multi-level design of manufacturing systems. The remainder of this chapter discusses manufacturing system design and simulation as well as the research problem and research objectives.

1.1 Manufacturing System Design

In **systems engineering**, system elements are designed for optimal system performance rather than optimal element performance. With respect to manufacturing system design, a systems approach calls for the division of a system design into sub-designs, and the specification of these sub-designs over multiple stages, where each subsequent stage represents a more detailed and complete specification. This process is illustrated symbolically in Figure 1.

Associated with each stage are sets of constraint criteria, which sub-designs must satisfy either individually or collectively, and sets of performance criteria, which are used to evaluate competing specifications. These criteria set guidelines, which within the view of the designers, are appropriate for the level

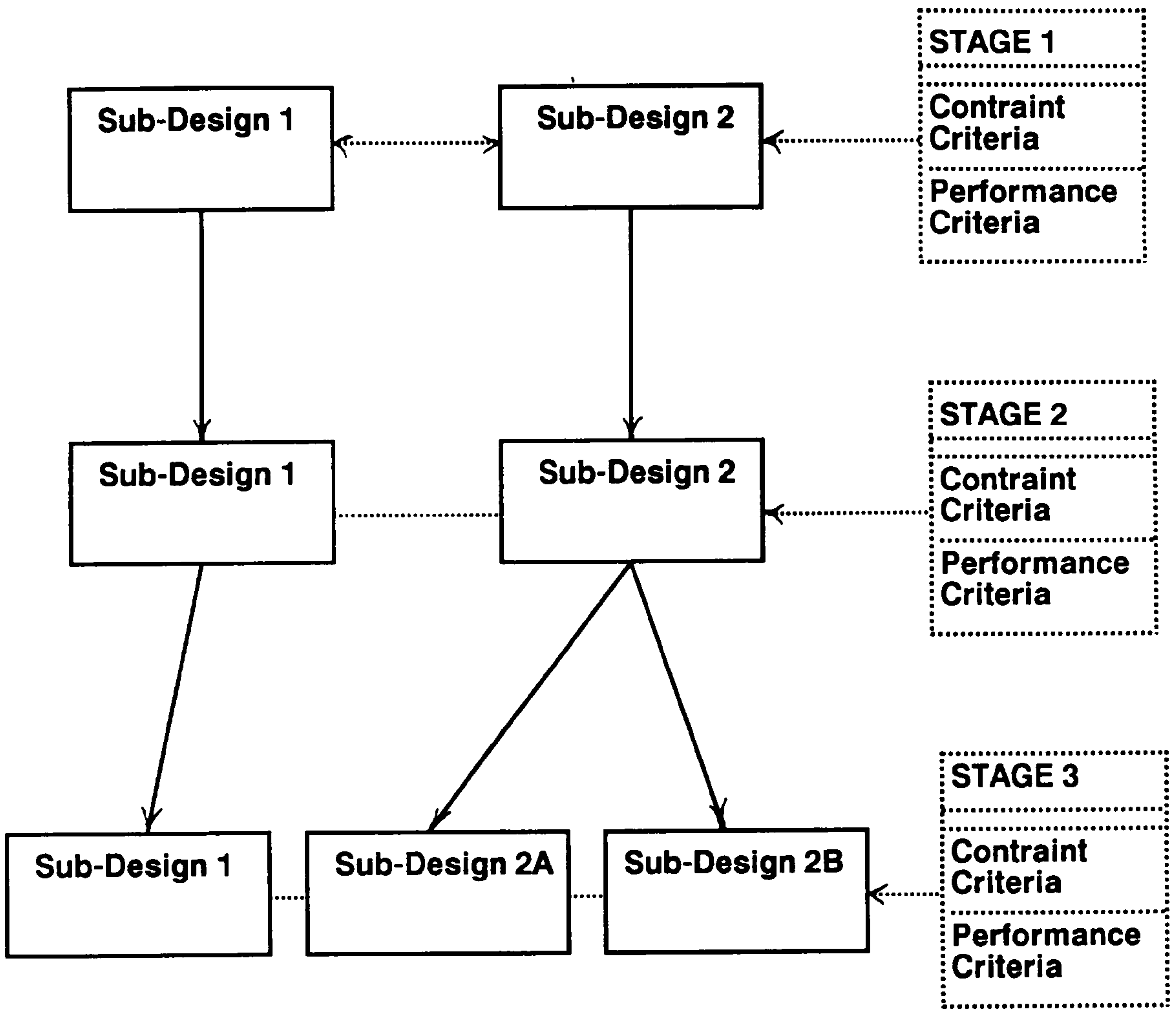


Figure 1 Multi-level Design Process

of specification. The stages at which criteria are used reflect their priority. The higher the priority, the earlier the criteria are applied.

In the initial stages of a multi-level design process, constraints are somewhat fuzzy, due to the uncertainty of design parameters such as volume, unit costs, and production rates. In addition, the selection of manufacturing system specifications are typically based on inaccurate evaluations due to the unavailability of accurate system data. In general, as a design specification becomes more complete, constraint and performance evaluation validity improve. Note that the initial stages of a design process are greatly facilitated if data from existing systems are available for extrapolation.

During a transition from one stage to the next, multiple alternatives are constructed for each sub-design. Data which influence their construction include constraint criteria and design specifications of interacting sub-designs at the new stage. The alternatives are evaluated with respect to the new performance criteria. If the best alternative satisfies the constraint criteria, and if its performance falls in-line with its performance from the previous stage, it is accepted as the new sub-design specification. However, if it fails to satisfy the constraint criteria, or its performance is significantly different than anticipated, then the sub-designs at the previous stage are re-designed and evaluated with respect to the new sub-design specification.

This process is repeated, until all sub-designs have been specified at the new stage. At the end of a transition, the new sub-design specifications may be evaluated collectively to insure that system constraints are met, and to verify system performance. Note that in many cases, a sub-design is divided into multiple sub-designs at a subsequent stage.

To evaluate sub-designs either individually or collectively, models are created and executed, and inferences are made with respect to their results. Model construction is typically the most time consuming of the three processes. This bottleneck is reduced considerably if models are created and manipulated in a modular fashion.

Modularity permits models to be assembled and executed in multiple configurations. Thus, once a sub-design model has been created, it can be used repeatedly in the assembly of other models. This is important, because sub-designs are repeatedly evaluated in collective groups throughout a multi-level design. Without modularity, the creation of aggregate design models would require their construction from scratch.

A barrier to modularity is the requirement that models be capable of heterogeneous assembly. Heterogeneous assembly is required when sub-designs, which are specified heterogeneously, are evaluated collectively. Currently, there are three methods of systems analysis that are applied to manufacturing system design. Method selection is greatly influenced by the level of design specification. In all but extreme cases, the methods are fairly exclusive. Thus the assembly of heterogeneous models poses a significant problem.

A partial solution is the modularization of the methods with respect to their individual problem domains. This will permit the modular construction of sub-design models, which utilize the same analysis technique. While it will not insure modularity throughout a multi-level design process, it will insure modularity for many consecutive design and evaluation stages. This dissertation investigates the modularization of computer simulation. The reasons for the selection of this technique are discussed in the following review of systems analysis.

1.2 Systems Analysis

Systems analysis is the process of constructing a model of a system, and performing experiments with the model to provide insight as to the operation of the system [44]. Today, it is regularly performed on a large number and assortment of existing manufacturing systems and designs of manufacturing systems as a means of process improvement.

There are three methods commonly used for the systems analysis of manufacturing systems. They are analytic, iconic, and computer simulation modeling.

Analytic modeling involves the creation of mathematical models to represent system behavior and can be both prescriptive (i.e., linear programming models) and descriptive (i.e., queuing networks) in nature. Examples include the use of queuing networks to analyze the aggregate behavior of manufacturing systems [47] and finite element analysis to estimate the deformation of blanks during forging operations [8,43]. The advantage of analytic modeling is the exactness of its solutions. With respect to modularization, its disadvantage is its utilization of a large variety of incompatible solution techniques.

Iconic modeling involves the creation of scaled, physical representations of actual systems. Its use provides an excellent source of visual, logical, and geometrical information. As a result, it is reported by Wysk, Ghosh, and Wu [53] to be excellent for manufacturing education. Unfortunately, iconic modeling is not very well adapted for repetitive manufacturing system analysis due to the high cost of model construction, difficulty with data extraction, and its inability to reflect true system performance characteristics.

Digital computer simulation modeling is the most widely used of the modeling methodologies for manufacturing system analysis [42]. Its popularity is a result of its flexibility, which allows it to describe a large range of manufacturing system behavior. With respect to modularization, computer simulation has the advantage that it uses a small set of compatible mechanisms. Its disadvantages are the difficulties associated with output analysis and the cost and expertise needed for model creation [5].

Of the three methodologies just described, computer simulation shows the greatest potential for modularization. In addition, due to its relatively large problem domain, its modularization should have the largest impact on manufacturing system design.

1.3 The Importance of Modularity in Computer Modeling

Balci and Nance [6] state that a computer simulation analysis consists of the following steps:

1. Problem Identification: definition of the problem to be corrected.

2. Objective Specification: determination of the problem analysis objectives.

3. System Definition: determination of system boundaries and restrictions.

4. Model Formulation: construction of the system model.

5. Model Translation: translation of the model into a form acceptable to the computer.

6. Model Validation and Program Verification: determination of the validity of the model with respect to the observed operation of the system.

7. Model Experimentation: manipulation and execution of the computer model to obtain output values.

8. Model Interpretation and Application: drawing inferences about the real system behavior from the model output.

Steps 4 through 8 tend to be an iterative process [6], where the simulation analyst reformulates and experiments with the system model in order to make comparisons between competing alternative systems.

Balci and Nance [5] state, that on average, 70 to 90 percent of the cost of a simulation study is spent on the iterative process of model formulation, translation, and validation. They further state that a large factor influencing this cost is computer model reusability, which in turn is a direct function of computer model modularity. This opinion appears to be held by others in the computer modeling community such as Zeigler and Oren [57] and Medeiros and Sadowski [32].

Model modularity relates to the ability of a model to be divided into sub-models, each of which can be removed or replaced with minimal effect on its neighboring sub-models. The importance of modularity pertains to both initial model construction and alteration.

With respect to initial model development, modular construction permits the rapid development of a valid simulation model by allowing a modeler to divide a system under study into sub-systems (objects), develop and validate small simulation models to represent these objects, and then link these models together to create an aggregate model. In general, the modular construction of software is considered to be a basic tenet of good software development.

With respect to reusability, model modularity permits a system modeler to alter an existing model through the modification or replacement of a set number of sub-models, allowing the majority of the existing model to remain intact. Without modularity, model alteration requires the modification of larger percentages of simulation code.

1.4 Uni-level and Multi-level Simulations

A simulation model of a manufacturing system is considered uni-level if all objects of equivalent type within the system are modeled at the same level of detail. For example, consider parts flowing through a workcell containing a pallet, two machine tools, and a robot. If the machine tools, robot, and pallet are modeled at the same level of detail then the model is considered uni-level.

This is demonstrated in Figure 2, where the system is represented with a discrete event simulation model utilizing a queuing network analogy. In this model, the machine tools and robot are modeled as servers, the pallet as a buffer, and the parts as customers. The attributes of the machine tools and robots are their busy/idle status. The attributes of the pallet are its capacity and the number of parts that it is currently holds. The attributes of the parts are their symbolic state and the identity of the pallet or server which they are currently located at. During simulation, the parts flow between the entities, demanding their service or storage capacity.

A simulation model of a manufacturing system is considered multi-level if objects of equivalent type are not modeled at the same level of detail. This is

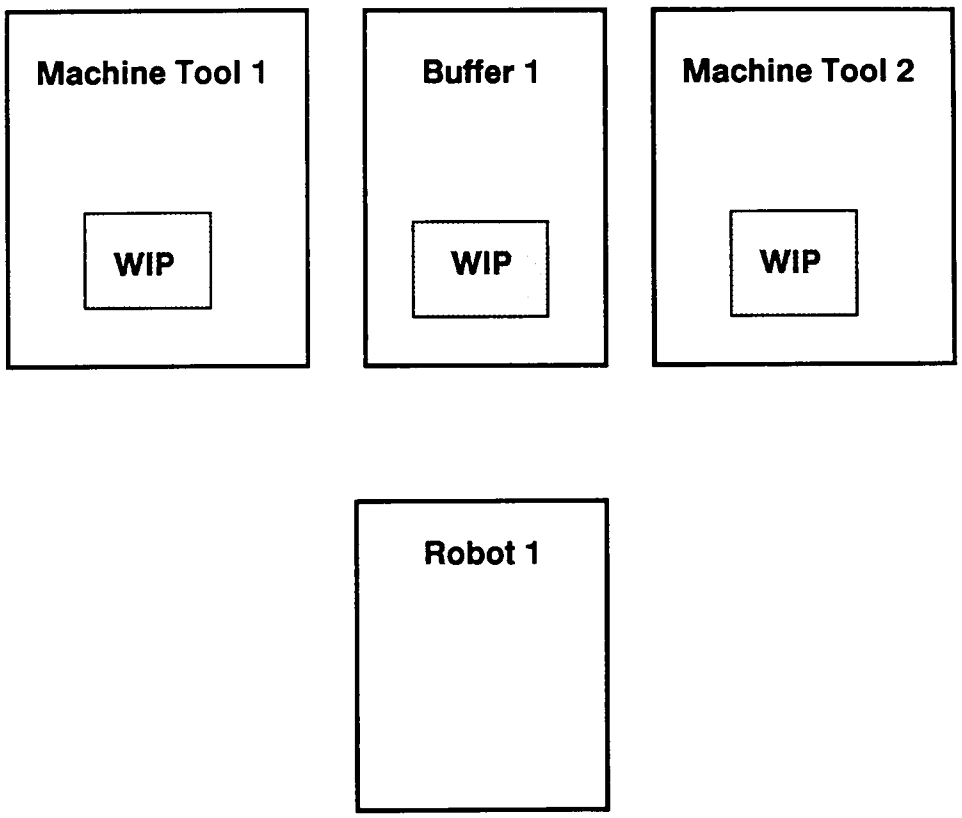


Figure 2 Uni-Level Representation of a Work Cell

demonstrated with the example system in Figure 3. Here the discrete event simulation models of the robot and the part that it interacts with have been replaced with continuous simulation/geometric models. The new models are used to represent their kinematic and geometric behavior. During simulation, the two sets of models work together in a hybrid fashion. Of interest is the dynamic nature of the part model attributes. When interacting with the machine tool or buffer models, the part models utilize symbolic attributes. When interacting with the robot model, they utilize kinematic and geometric attributes.

The use of a uni-level simulation model allows a modeler to simulate both aggregate system behavior and the interactions among objects within a system. By its nature, a uni-level simulation model does not focus attention on any one object, since they are all modeled equally. A review of the literature suggests that this type of model is commonly used in the analysis of existing manufacturing systems or designs of manufacturing systems in which all sub-systems have been designed to the same level of detail. With respect to the multi-level design process, uni-level simulation models are applicable for the collective analysis of a broad range of homogeneous sub-designs.

Like a uni-level simulation model, the use of a multi-level simulation model allows a modeler to simulate both aggregate system behavior and the interactions among objects within a system. But, in addition, a multi-level model permits the modeler to focus on the behavior of a fixed group of objects. As a result, a multi-level simulation model conceptually represents multiple systems.

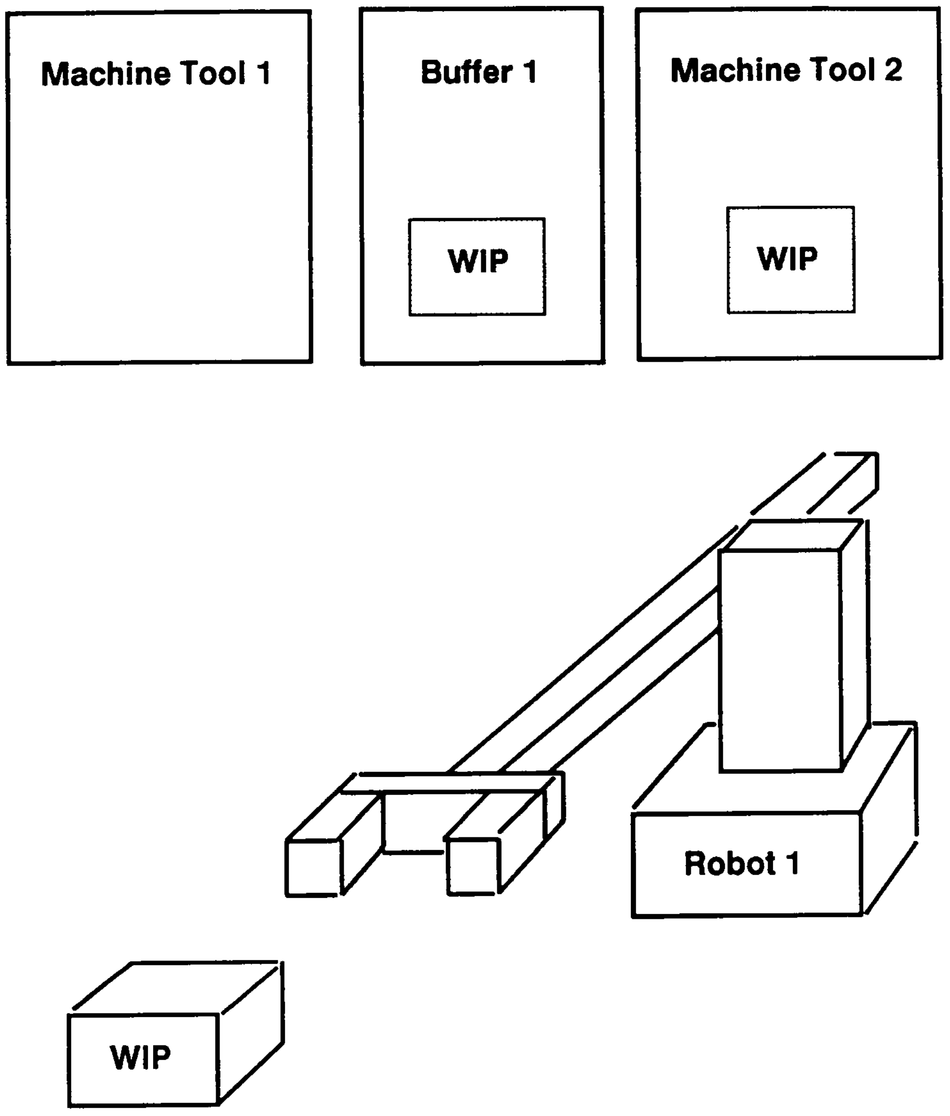


Figure 3 Multi-Level Representation of a Work Cell

This concept, as applied to our example multi-level simulation model, is illustrated in Figure 4. Here the model represents three systems. System 1 consists of the machine tools and the buffer along with the parts that flow through them. This system is represented via a discrete event/queuing network model. System 2 consists of the robot and the part that it interacts with. It is represented via a kinematic/geometric model. System 3 is the work cell itself, and it is represented by the aggregate multi-level model.

During execution of the multi-level simulation model, Systems 1 and 2 provide each other with inputs along with inputs received from the aggregate model. In turn, they provide the aggregate model with outputs.

The composite nature of multi-level simulation models makes them well-suited for the collective analysis of heterogeneous sub-designs. Through proper experimentation, they allow system designers to analyze not only the potential performances of detailed sub-systems, but also the potential effects of sub-systems on aggregate system performance as well as vice versa. This allows a system designers to compare alternative detailed designs of sub-systems without having to model the aggregate system at the same level of detail.

1.5 Research Problem

In an ideal world, manufacturing system designers could rapidly construct simulation models of sub-designs and store them in a model-base. To evaluate

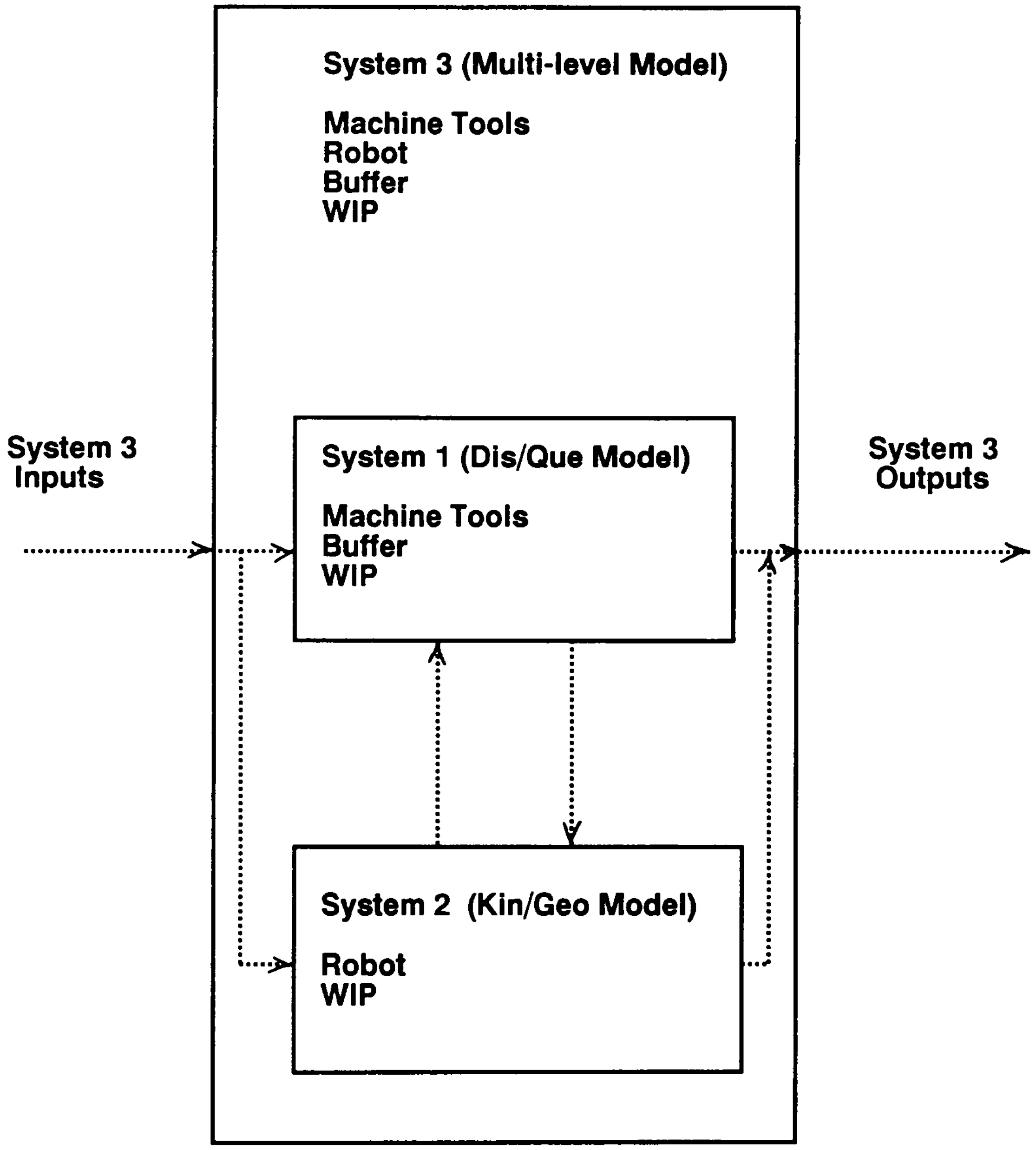


Figure 4 Three-System View of Work Cell

sub-designs either independently or collectively, designers could select models from the model base, assemble them in any uni-level or multi-level combination, and then execute them. Unfortunately, very few simulation frameworks support the modular construction of simulation models, and no simulation framework utilizes all of the mechanisms needed to support multi-level modeling.

The execution of an assembly of independent simulation models requires extra overhead for their synchronization, resulting in greater execution time. Over the past thirty years, the costs of executing simulation models has exceeded the costs of developing them [4]. As a result, the modeling community has emphasized execution efficiency rather than development efficiency. Consequently, manufacturing system modeling has been dominated by fast executing, non-modular simulation frameworks. Ironically, the relationship between model execution and development has changed drastically, due to improvements in computer technology and large increases in the cost of skilled labor.

To support multi-level modeling, a simulation framework must utilize the simulation methods and time flow mechanisms found in both discrete event and continuous simulation modeling, maintain a generalized time flow mechanism, and handle dynamic attribute sets. For example, in the multi-level simulation model presented in Figure 3, the discrete event representation of the machine tools requires the execution of event functions and a variable increment time flow mechanism, while the continuous/geometric representation of the robot requires the execution of attribute description functions, conditional activities, and a fixed increment time flow mechanism. To maintain synchronization within the composite simulation, a time flow mechanism for the aggregate model must be executed. This time flow mechanism must not only work in conjunction with the time flow mechanisms of the constituent models, but it must also be independent of their type. The need for dynamic attribute sets was discussed earlier. Currently, no simulation framework described in the literature has these characteristics.

1.6 Research Objective

The objective of this dissertation is to develop a simulation framework which will support the modular construction of uni-level and multi-level simulation models of manufacturing systems.

To demonstrate its feasibility, the framework will be applied to the simulation of a hypothetical manufacturing system. Note that a solution alternative (automatic model generation) other than the development of a new framework has been considered. This alternative, along with the reasons why it was not pursued, are discussed in chapters 2 and 3.

1.7

~~1.6~~ Significance of Research

The multi-level design approach shows promise as an effective means of manufacturing system design. The development of a simulation framework which can support the modular construction of uni-level and multi-level simulation models will eliminate a large barrier to its common implementation.

In stand alone form, the use of the framework should dramatically reduce the time needed to develop and alter simulation models. When used as a basis for automatic program generation (discussed in chapters 2 and 3), the impact of the framework should be amplified to an even greater extent.

The development of the framework is just the beginning of a long research effort toward the development of a new generation of computer-aided manufacturing system design environments. It is anticipated that these environments will support the multi-level design process by aiding designers with computer-assisted specification, automatic model generation, automatic model management, and knowledge-based simulation analysis. The implementation of these environments will dramatically change the manner in which manufacturing system design is conducted.

1.8 Summary of Dissertation Contents

This chapter has provided an introduction to the research topic. The remainder of this dissertation details the development of the simulation framework. Chapter 2 provides a literature review on the various computer modeling techniques applied to manufacturing system analysis, the historical attempts of bypassing the modularity problem, and simulation frameworks which have been developed to support the modular, multi-level design process. Chapter 3 details the research methodology, describing why the development of a new simulation framework was chosen as the problem solution. In addition, the performance requirements for the framework are discussed along with software and hardware considerations and the methodology for demonstrating the framework. Chapter 4 provides a detailed explanation of the simulation framework and how it can be applied in single process and parallel process applications. Chapter 5 describes a model demonstration environment that was created to demonstrate the simulation framework. Finally, chapter 6 summarizes the dissertation, provides conclusions, and discusses future research.

2 LITERATURE REVIEW

2.0 Introduction

This literature review discusses the applications and mechanics of computer modeling techniques commonly used in the analysis of manufacturing systems. These techniques include discrete event simulation, continuous simulation, geometric modeling, and graphical animation. The review also discusses the role of automatic model generation as a historical solution to the lack of modularity of conventional simulation frameworks, as well as the simulation frameworks which have been developed to support the multi-level design process.

2.1 Discrete Event Simulation

Discrete event simulation is a computer modeling technique used to model the event oriented behavior of systems over the simulated passage of time. The following is a discussion on how discrete event modeling is normally applied to manufacturing system analysis as well as a description of discrete event simulation mechanics.

2.1.0 Application in Manufacturing System Analysis

Discrete event simulation models are used extensively to analyze the queuing dynamics of manufacturing systems. At a high level of abstraction, manufacturing systems are normally modeled as queuing networks [42]. Conceptually, resources within the system are modeled as servers while entities such as parts are modeled as customers. During simulation, parts conceptually flow through the simulated system, queuing behind resources and demanding their services. Events normally simulated are the arrival of parts into the manufacturing system, the queuing of parts behind resources, the beginning and completion of services by resources, and the removal of parts from the system.

As progressively less abstract models are used to represent manufacturing systems, greater numbers of system resources are modeled along with their more complex interactions. This increase in detail results in queuing network simulation models of greater complexity.

For example, consider a hypothetical manufacturing system containing two work cells, each separated by physical buffers and each consisting of a robot and a machine tool. Conceptually, this system could be modeled as shown in Figure 5. Here the work cells are treated as servers and the buffers as storage. The interactions modeled are those between the work cells, buffers, and parts.

At a lower level of abstraction, the manufacturing system could be modeled as shown in Figure 6. Here the robots and machine tools within the work cells are treated as servers. The interactions modeled are those between the machine tools, robots, buffers, and parts. As progressively less abstract models are applied to this system, elements such as grippers, cutters, sensors, etc. are represented along with their associated interactions.

This queuing network analogy has been utilized by many commercial, manufacturing-oriented, discrete event simulation languages and environments. These simulation tools provide modeling constructs which allow a user to create models quickly through the construction of queuing networks. Examples include AutoMod [16], FACTOR [33], ModelMaster [9], MAST [29], PCModel [51], SIMAN [41], SIMFACTORY [48], SLAM II [42], STAR*CELL[46], WITNESS [22], and XCELL [13].

2.1.1 Discrete Event Simulation Mechanics

A discrete event simulation model is used to represent the changes of state of a system over discrete points in time [42]. The term **world view** is often associated with the implementation of discrete event models. A world view

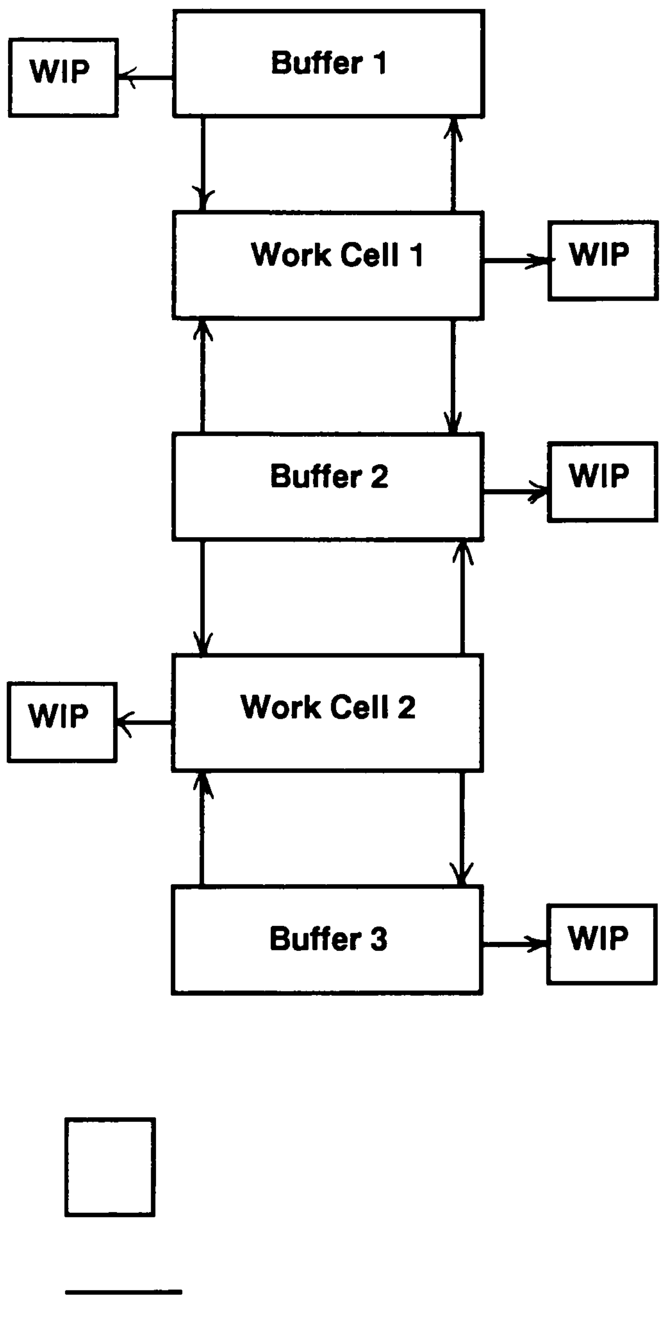
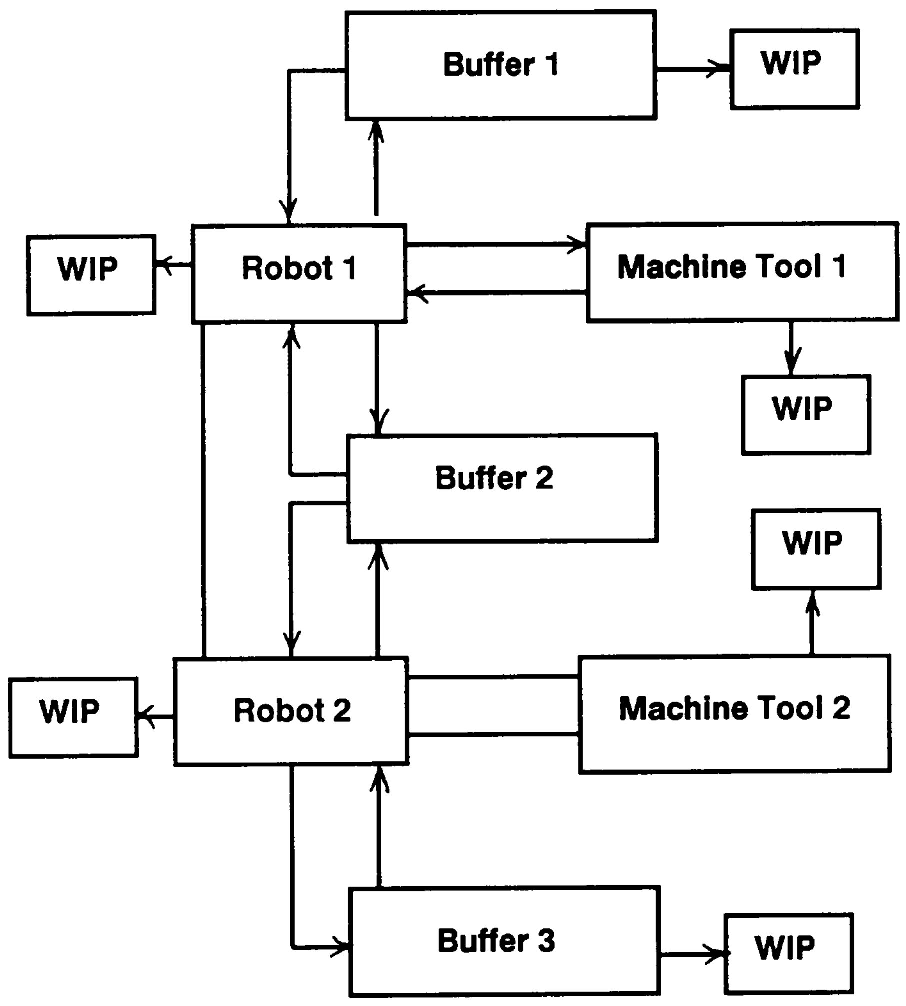


Figure 5 Queuing Network Analogy



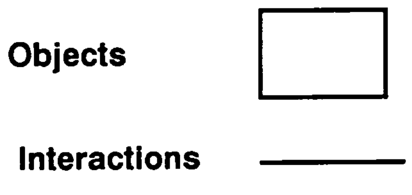


Figure 6 Complex Queuing Network Analogy

(also called a framework) is a description of the mechanisms used to execute a simulation. Currently there are many world views associated with the implementation of discrete event models. The most commonly used are **event scheduling**, **activity scanning**, the **three phased approach**, and **process interaction**. In addition a fifth, **DEVS** (Discrete EVent System) specification has recently gained attention as an approach which takes advantage of the **object oriented programming paradigm**. These five world views are described next. But first some additional computer modeling concepts need to be discussed. Note that much of the following discussion is taken from Balci [4].

A system can be described in terms of **objects**, **attributes**, **events**, **activities**, and **processes**. An object denotes an element of interest in the system. An attribute denotes a property of an object, a property of the system, or conveys information about an aspect of an object or the system.

An event is anything that causes a change in the state of an object and/or the system. The state of an object is defined by the values of all attributes of that object at a particular instant of time. The state of a system is defined by the values of system attributes and object attributes at a particular instant of time. An activity is what transforms the state of an object over a period of time. An activity is initiated by the occurrence of an event and is ended by the occurrence of another event. A process is a sequence of activities or events ordered with respect to time.

As an illustrative example, take the discrete event representation of robot 1, machine tool 1, buffer 1, and buffer 2 in Figure 6. The objects of interest are the robot, machine tool, and buffers as well as the parts that flow through the sub-system. An attribute of the robot and machine tool is their busy/idle status. An attribute of the buffers is the current number of parts that they hold. Attributes of the parts are their time of arrival into the system, their time of departure from the system, and current volume. An attribute of the system is the current number of parts in the system. Examples of events and activities associated with this system are shown in TABLEs 1 and 2 respectively.

The time flow mechanism of a discrete event model relates to how simulated time is advanced during model execution. There are two types of mechanisms, **fixed increment** and **variable increment** (also called **next event**).

In a simulation utilizing a fixed increment scheme, the simulation clock is updated by the addition of a fixed increment of time. This fixed increment is pre-determined by the modeler before simulation execution. The proper increment size is dependent upon the behavior of the system being modeled. In general, the increment size needs to be less than the smallest anticipated time period between state changes during model execution.

In a simulation utilizing a next event scheme, the simulation clock is updated to the time of the next event to occur in the executing simulation. This involves tracking all anticipated future events along with their associated firing times. In general, the next event scheme is considered to be more efficient with respect to actual simulation execution time than the fixed increment scheme. However in many modeling situations, the task of tracking future events can be

TABLE 1 Events for Example Discrete Event Simulation

Event	Description	Attributes Changed
E1	Arrival of Part at Buffer 1	Status of Part, Buffer 1, and System
E2	Robot 1 Grasps Part at Buffer 1	Status of Buffer 1 and Robot 1
E3	Robot 1 Releases Part at Machine Tool 1	Status of Robot 1 and Machine Tool 1
E4	Machine Tool 1 Begins Processing Part	Status of Machine Tool 1
E5	Machine Tool 1 Finishes Processing Part	Status of Machine Tool 1 and Part
E6	Robot 1 Grasps Part at Machine Tool 1	Status of Robot 1 and Machine Tool 1
E7	Robot 1 Releases Part at Buffer 2	Status of Robot 1 and Buffer 2
E8	Part is Removed from Buffer 2	Status of Buffer 2 and System

TABLE 2 Activities for Example Discrete Event Simulation

Activity	Description	Beginning Event	Ending Event
A1	Robot 1 Waiting for Part Arrival	E1	E2
A2	Robot 1 Transferring Part to Machine Tool 1	E2	E3
A3	Machine Tool 1 Waiting for Part Arrival	E3	E4
A4	Machine Tool 1 Processing Part	E4	E5
A5	Robot 1 Waiting for Machine Tool 1 to Finish	E2	E5
A6	Robot 1 Transferring Part to Buffer 2	E6	E7
A7	System Waiting to Remove Part from Buffer 2	E7	E8

complicated and time consuming. In these circumstances, the fixed increment implementation is more efficient [5].

Discrete event simulation is not effective for modeling system behavior which changes continuously with respect to time. This is due to the difficulty and inefficiency of trying to model continuous behavior with an event oriented technique. Modeling system behavior such as this requires the use of continuous simulation.

2.1.1.0 Event Scheduling World View

Under the event scheduling framework, an event is the major focus of the system model. This is illustrated in Figure 7. Initialization includes the assignment of initial values to all system and object attributes, the setting of the clock to zero, and the initialization of the **event list**. Assuming that the simulation starts with the modeled system empty of parts, the event list is initialized by merging the first events to occur in the future. In the illustrative example, this is the arrival of parts into buffer 1.

Records are used to store event information. At a minimum, such information includes **event type** and the **future time of occurrence**. Note that event records are merged into the events list in order of increasing future occurrence time.

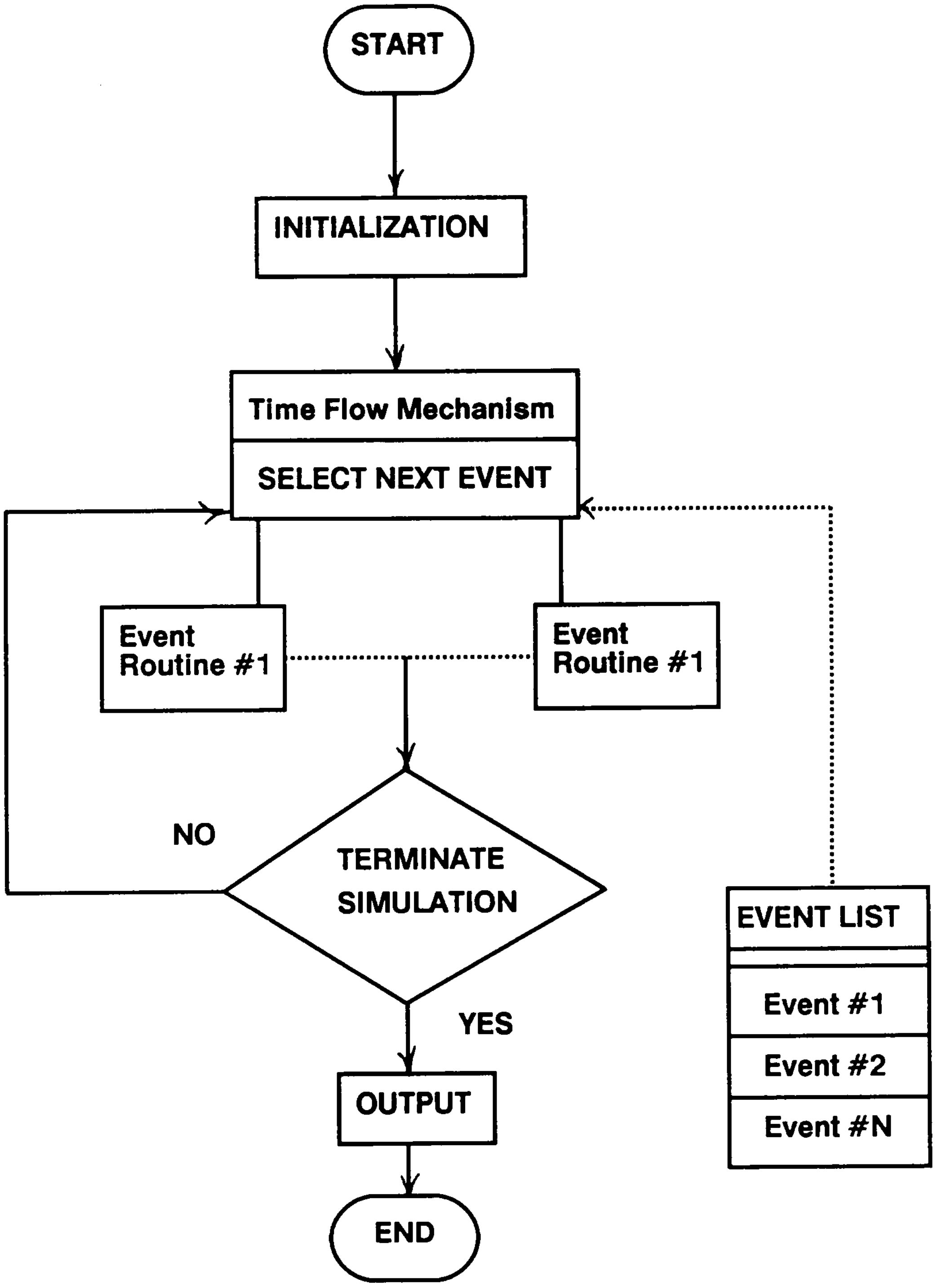


Figure 7 Event Scheduling World View

During simulation execution, the system clock is updated either using the fixed increment or variable increment method. When the clock is equal to the occurrence time of the first event on the events list, the function associated with the event is called, the event record is removed from the top of the events list, and the clock is again updated. Simulation continues until some preset condition for termination is met. At this time, numeric simulation results are transferred to some output medium.

An event function performs two duties. The first is to update some combination of object and system attributes. In the case of an arrival event in the example simulation, this entails creating a part record, initializing its attributes, incrementing the number of parts currently held by buffer 1, and incrementing the total number of parts in the system.

The second duty performed by an event function is to spawn any additional events that may be conditional with respect to its occurrence, and to merge them into the events list. For example, assuming that parts arriving into the illustrative system follow a particular arrival time probability distribution, then the firing of an arrival event spawns a new arrival event. The future occurrence time of this new arrival event is dependent upon the current simulated time and the probabilistic inter-arrival time.

A model created with the event scheduling world view is very efficient with respect to execution speed when the maximum number of events simultaneously appearing on the events list is small relative to the total number of generated events. This is especially true when the next event time flow mechanism is utilized. This efficiency is obtained by the direct linking of event functions, which results in a reduction of event condition checking. As a result, a model created under the event scheduling framework is characterized by a tight network of dependent simulation functions.

Unfortunately the efficiency gained by the use of the event scheduling framework comes at the expense of model modularity. The tight structure of an event scheduling model prohibits its alteration without a significant amount of code alteration in the simulation functions.

2.1.1.1 Activity Scanning World View

The activity scanning world view (also known as the **two-phased approach**) is a state-based approach to simulation modeling. Unlike the event scheduling approach, activities are the basic building blocks of a model.

Under this framework, activities are described in two parts. The first part is a condition or compounding of conditions which must be satisfied in order for the activity to take place. This condition may be time dependent or attribute dependent. The second part specifies which operations (including the update of object and system attributes) must be performed in order for an activity to completely fire.

As an example, take an activity that is utilized to simulate the arrival of parts into the example simulation. During execution, the condition that is checked is whether the current value of the clock is equal to the arrival time of the next part to enter the system. If this condition is met, then the activity fires and all of the attributes that are dependent upon a part arrival are updated. In addition, the time of the next arrival is computed, providing the basis for future firings of the activity.

Figure 8 illustrates the logic of the activity scanning approach. Initialization involves assigning values to attributes and spawning activities. Phase 1 involves the time scan. In the activity scanning approach, the fixed increment time flow mechanism is used. As a result, the clock is simply incremented by a fixed time step.

Phase 2 involves the activity scan. During this phase each activity is scanned to determine if it can fire. The sequential scanning of the activities is repeated until all activities fail to fire. Note that the activities are assigned priorities, and are scanned in order of priority. The reason for assigning priorities is that activities are typically dependent upon one another. As a result, activities with the least dependencies are assigned the highest priorities and activities with the greatest dependencies are assigned the lowest priorities. This is done to minimize the number of scans required for any given execution of phase 2. Simulation termination and output are conducted in the same fashion as the event scheduling world view.

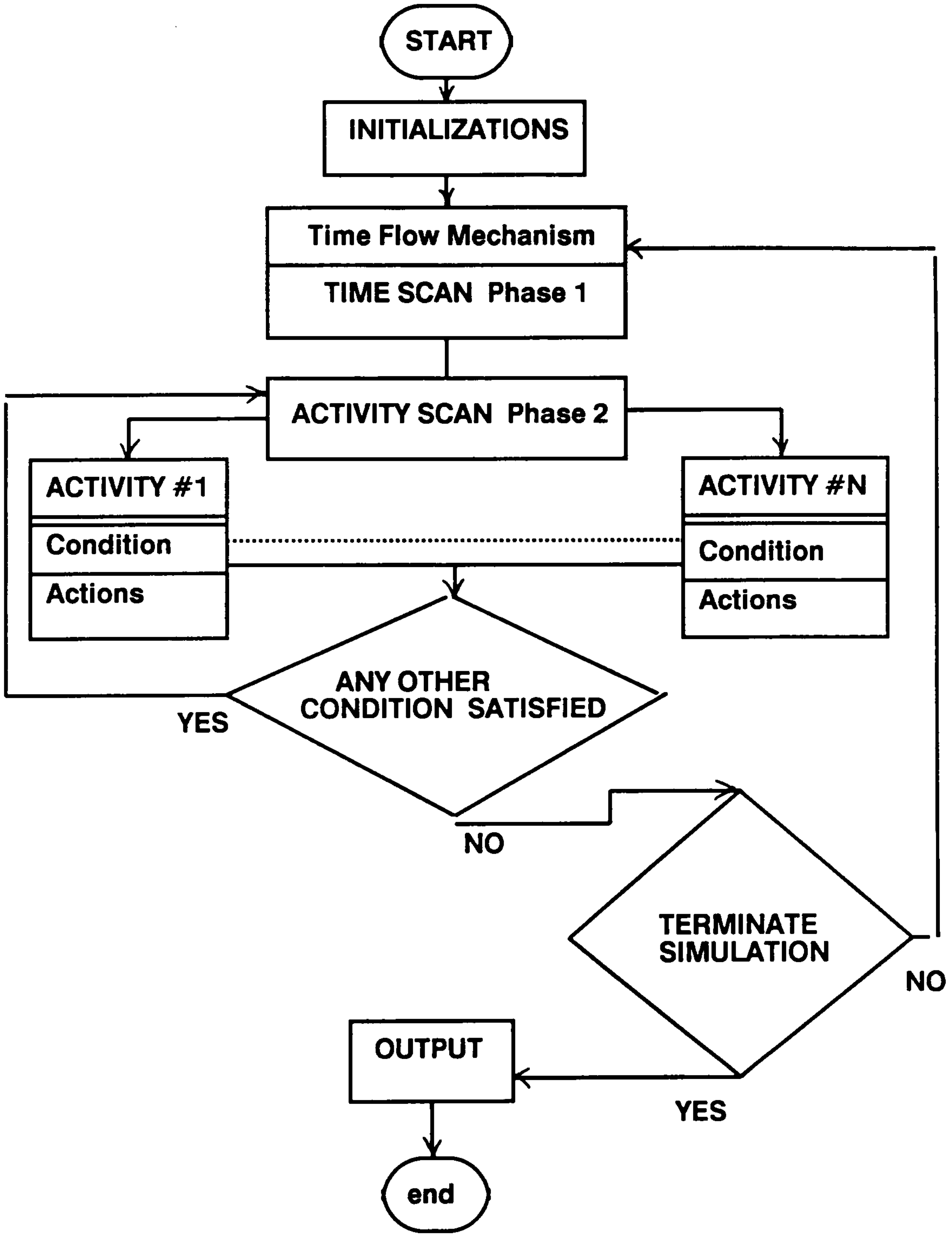


Figure 8 Activity Scanning World View

The activity scanning approach produces a simulation composed of independent modules waiting to be executed. Thus it is a highly modular approach of model construction. Unfortunately, due to its use of the fixed increment time flow mechanism and its need to repeatedly scan activities during phase 2 execution, it is slower in execution relative to the other world views.

2.1.1.2 Three Phase World View

The three phase approach attempts to remedy the inefficiency of the activity scanning approach while still maintaining its modular structure. It does so by combining features of both activity scanning and event scheduling. Events and activities are the two basic building blocks of this approach. However, events are labeled as activities of duration zero.

As a result, activities are classified as either phase 2 activities or phase 3 activities. Phase 2 activities are strictly time dependent with duration zero. Phase 3 activities are conditional or co-operative activities that represent state changes dependent upon the satisfaction of specific conditions.

The logic of the three phase approach is illustrated in Figure 9. Like the activity scanning approach, phase 1 is the time scan. However with this approach, the variable increment time flow mechanism is used. As a result, the clock is always updated to the time of the most imminent phase 2 activity. Phase 2 involves the scan of phase 2 activities, determining which activities have

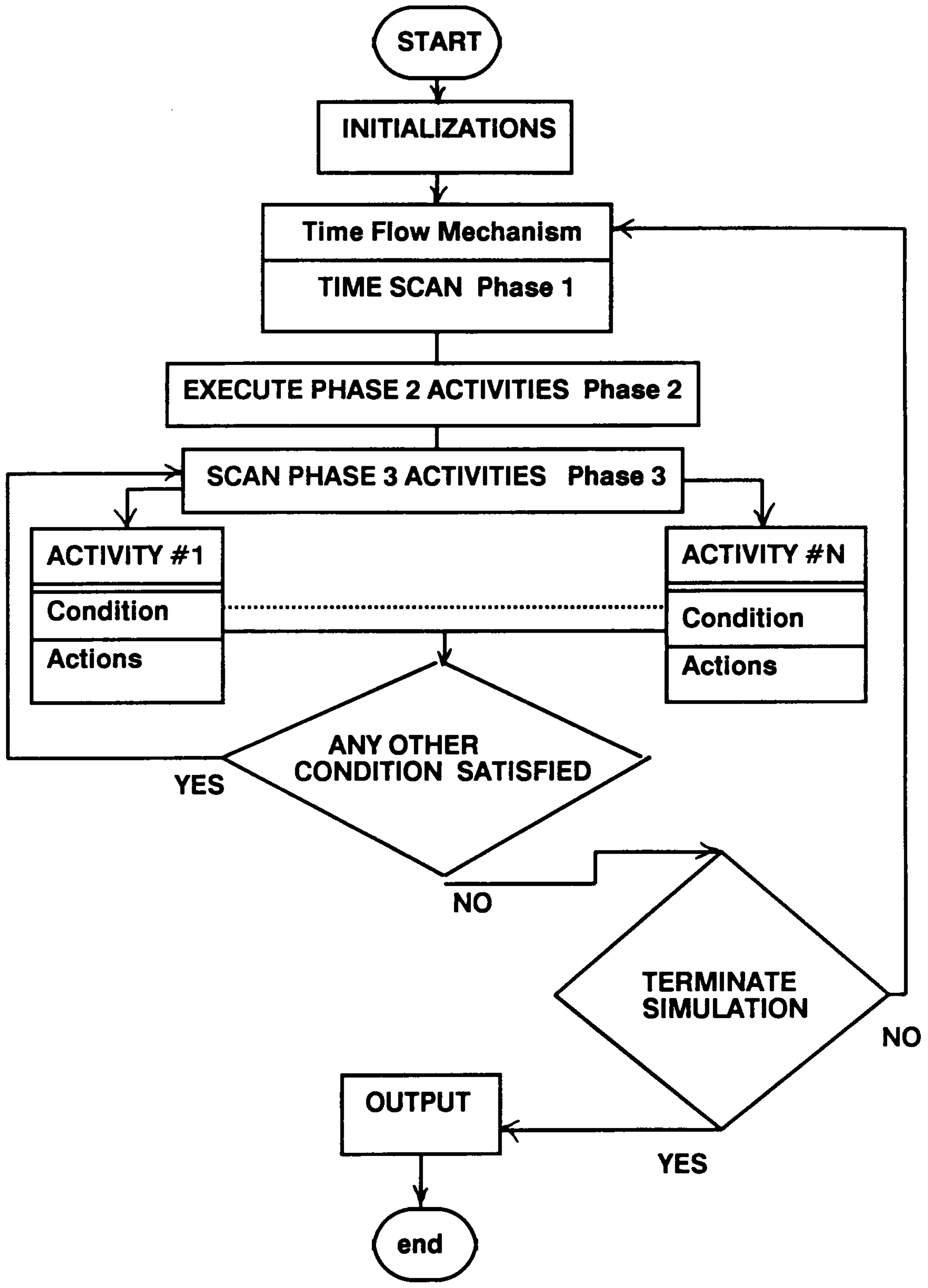


Figure 9 Three Phase World View

occurrence times equal to the value of the clock, and firing those activities. Phase 3 is conducted in identical fashion to phase 2 of the activity scanning approach, where the conditional activities are repeatedly scanned until all fail to fire. Note that simulation initialization, termination, and output are conducted in a fashion identical to that of the activity scanning approach. These modifications to the activity scanning approach make the three phase approach much more efficient with no loss of modularity.

It should be noted that neither the activity scanning nor the three phased approach are widely used in the United States, even though they are extremely popular in Europe [54]. The reason for this may be that during the late 60s and early 70s, when discrete event simulation was growing in popularity as an analysis tool, computer processing time was at a premium. Thus only the most efficient frameworks with respect to computer time were pursued. This has lead to the comercial popularity of general purpose languages such as SLAM II and SIMAN, which utilize the world view described next.

2.1.1.3 Process Interaction World View

Using the process interaction world view, a modeler describes the life cycle of an object which moves through and interacts with processes of the system under study. Object and process descriptions constitute the main body of this approach.

Objects are classified as either **static** or **dynamic**. A static object is one which does not logically move. It can have many attributes, but at the minimum it must have an **identifier**. A dynamic object is one which comes into the model, logically moves through some processes, and leaves the model. It too can have many attributes, but at the minimum, it must have attributes for **identification, move time, current location, next location**, and **priority level**.

In the example simulation, the robot, machine tool, and buffers are considered static objects while the parts flowing through the system are considered dynamic objects. Note how nicely this framework fits into the queuing network conceptualization of manufacturing systems. This is one of the primary reasons why it is so widely used to model manufacturing systems.

Processes are strings of events and/or activities. In the process interaction approach, processes are represented by **current objects list(s)**. During simulation, dynamic objects travel through these current objects lists, firing off the functions associated with the activities and events.

Along with the current objects lists, there is also a single **future objects list**. This list is used to track future move times in the same fashion as the event list tracks future events in the event scheduling approach.

Figure 10 shows the logic behind the process interaction world view. There are four phases of simulation logic: initialization, clock update, process scan, and output. During the initialization phase, system and static object attributes are given values, and dynamic objects are created, initialized, and placed on the

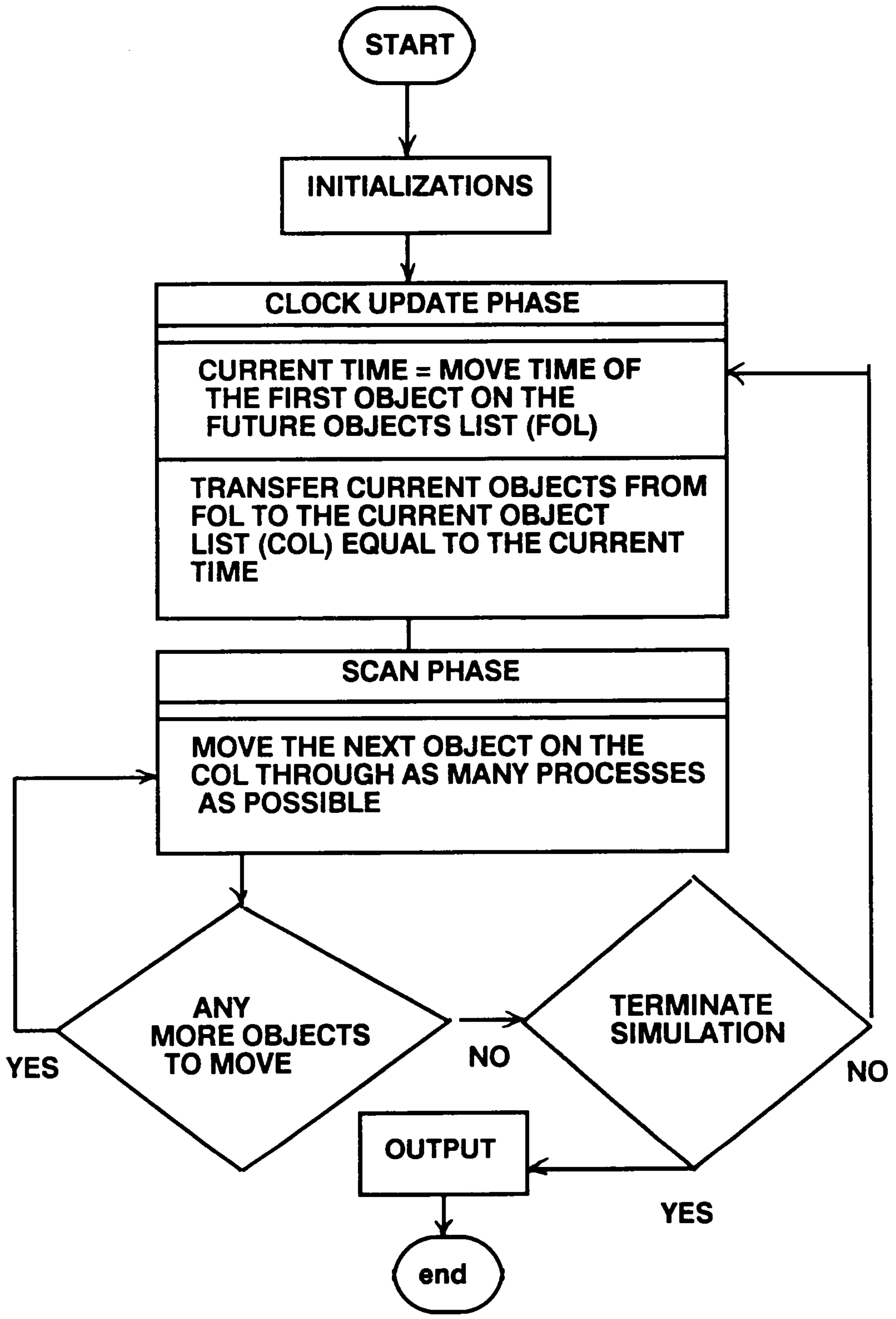

START
INITIALIZATIONS
CLOCK UPDATE PHASE
CURRENT TIME = MOVE TIME OF THE FIRST OBJECT ON THE FUTURE OBJECTS LIST (FOL)
TRANSFER CURRENT OBJECTS FROM FOL TO THE CURRENT OBJECT LIST (COL) EQUAL TO THE CURRENT TIME
SCAN PHASE
MOVE THE NEXT OBJECT ON THE COL THROUGH AS MANY PROCESSES AS POSSIBLE
ANY MORE OBJECTS TO MOVE
YES
NO
TERMINATE SIMULATION
NO
YES
OUTPUT
end

Figure 10 Process Interaction World View

future objects list in increasing order of move times. In the example simulation, these dynamic objects relate to the parts scheduled to arrive at buffer 1.

During the clock update phase, the clock is updated to the move time of the object on top of the future objects list. All dynamic objects in the future objects list which have move times equal to the value of the updated clock are placed on one of the current objects lists and queued in order of priority. Where the objects are placed is dependent upon their next location attribute. Once moved, their move time attribute takes on a value of "**As soon as possible.**"

Once all possible dynamic object moves from the future objects list to the current objects lists have taken place, the simulation goes into the process scan phase. In this phase, each dynamic object is moved one by one through as many processes (current objects lists) as possible. Each move through an activity or event in a process results in the firing of the function associated with that activity or event. Once a dynamic object is set in motion, only one of the following can stop it:

1. the object faces an unsatisfied condition,
2. the object is deliberately delayed,
3. the object dies or leaves the system, or
4. the object is deliberately stopped for some reason.

Note that when an object is delayed, its next location is assigned the identifier of the subsequent activity or process, its move time is assigned the value of the clock plus the delay time, and it is removed from a current objects list and merged into the future objects list.

The current objects lists are repeatedly scanned until no object moves are possible. At this time, the simulation goes back into the clock update phase. The need for repeated scans is due to dependencies among activities and events in the current objects lists. Simulation continues between these two phases until the conditions for termination are met. At this time the simulation transfers data to an output medium.

The process interaction world view provides an excellent technique for creating fast executing models, which are more modular than those created with the event scheduling world view. Modularity is gained by the reduction of the dependencies among event functions while speed is lost in some circumstances due to the introduction of a scan phase. However the process interaction world view still restricts modularity due to the physical links which must exist between and within current object lists. As a result, it does not provide the same degree of modularity as either the activty scanning or the three phase approaches.

2.1.1.4 Discrete Event System Specification World View

The discrete event system specification world view, DEVS, was introduced by Zeigler and Concepcion between 1986 and 1987 [12,55,56]. It was designed to take advantage of the object oriented programming paradigm. To provide a better explanation of the structure of DEVS, the characteristics of object oriented programming are discussed first.

2.1.1.4.0 Object Oriented Programming

Object oriented programming [39] is based on the concepts of **classes**, **processes**, and **message passing**. A class describes a set of instance variables, all potentially of different data types (integers, floating point, etc), a set of **methods** which operate on those variables, a set of class variables which act as global variables to all objects of this class type, and a pointer to a super-class. This is illustrated symbolically in Figure 11.

A class can be established relative to another class. In Figure 12, class A is a super-class of classes B, C, and D. Class C is a super-class of class D. Classes B, C, and D are all sub-classes of class A. Being a sub-class permits inheritance of all of the class variables and methods of a super-class.

An object (not to be confused with the definition given in section 2.1.1) is an instance of a class. As a result of being an instance, an object is allocated memory for storage of instance variables (those of its class and all sub-classes) and is given access to the methods of its class, sub-classes, and super-classes.

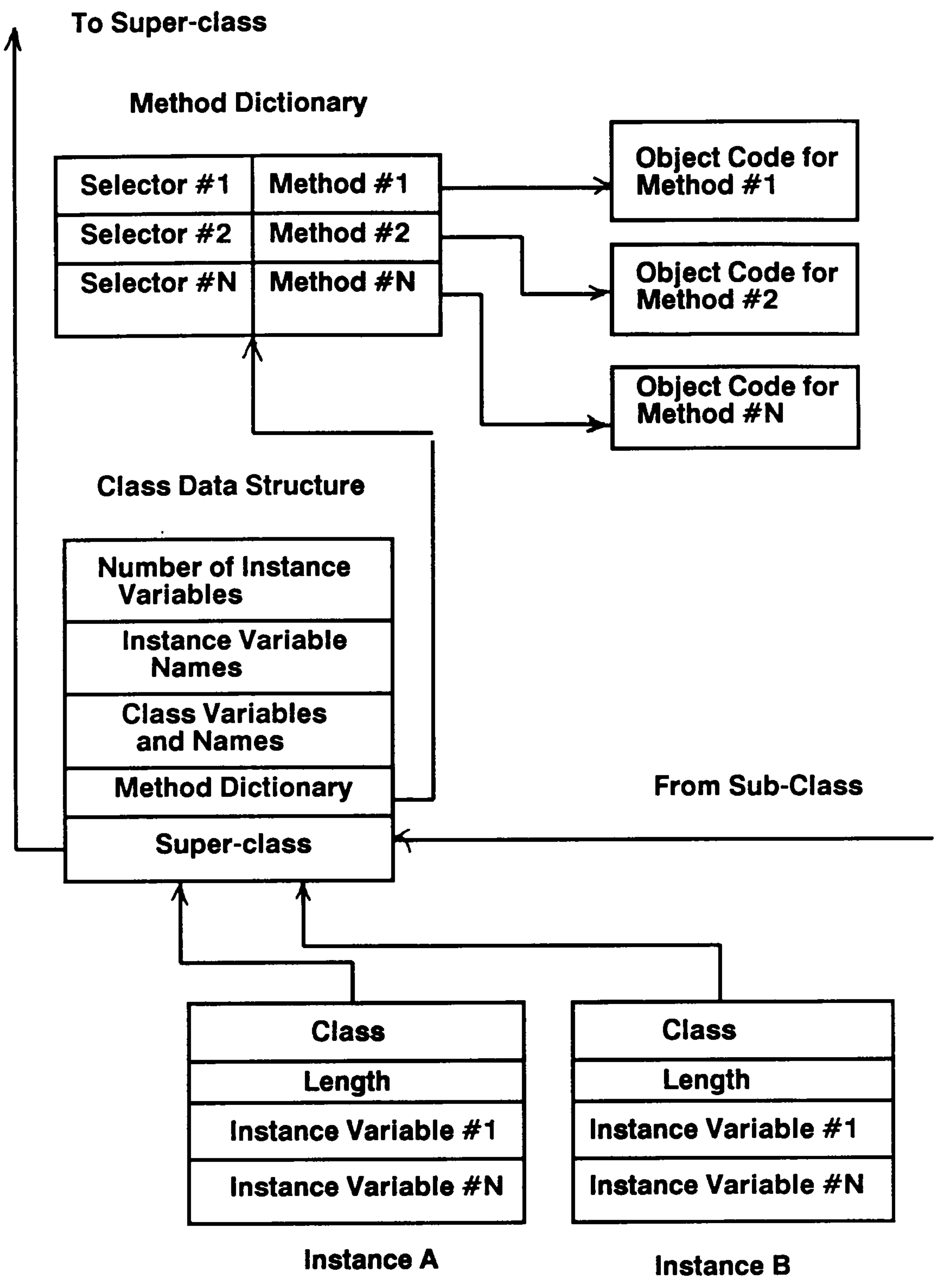


Figure 11 Object Oriented Programming Paradigm

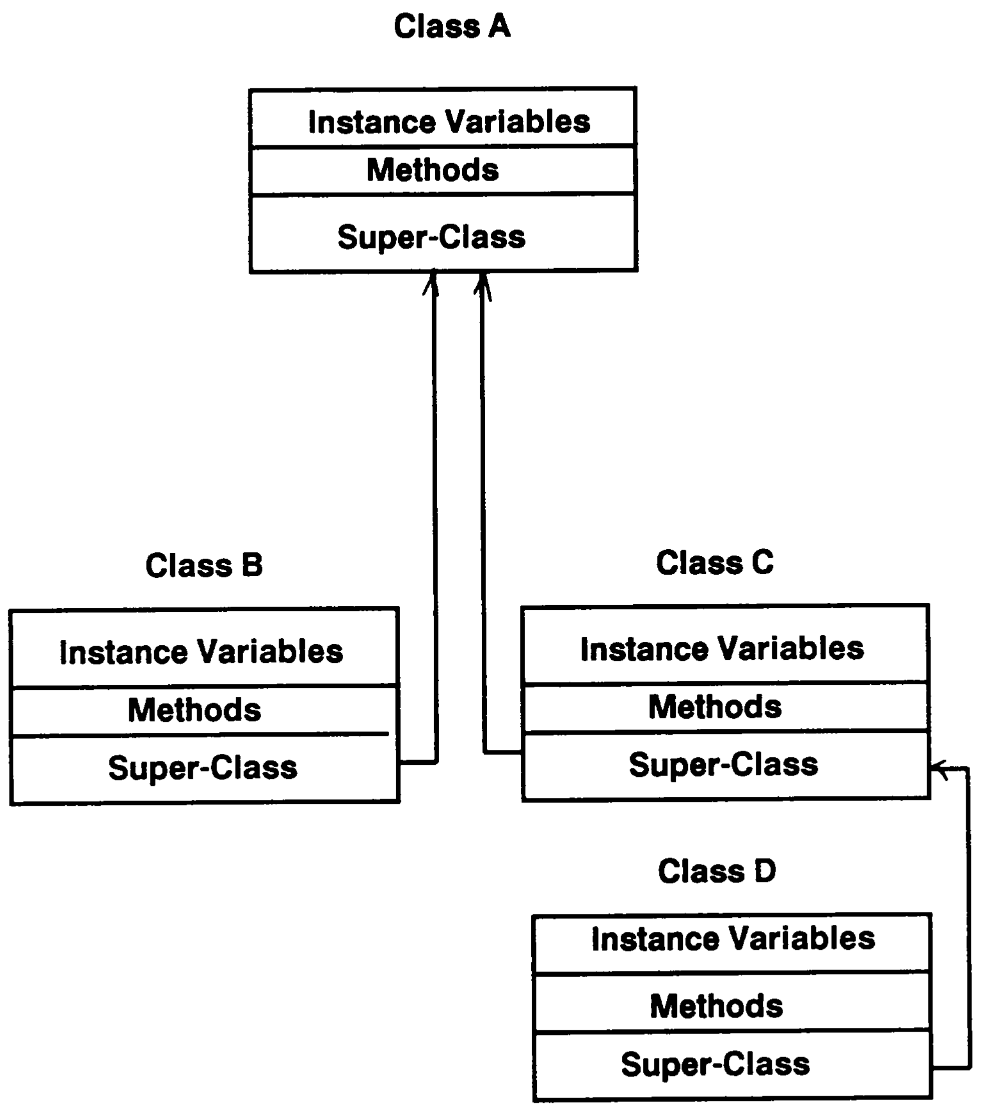


Figure 12 Class Structure Example

A process is a segment of executable code residing in RAM (random access memory) of a computer. In a single process system, there may be many processes residing in RAM memory at a time, but the computer CPU (central processing unit) will only execute these processes one at a time. In a multi-processing system, multiple processes may also exist in computer memory, but unlike the single process system, the CPU executes them some what simultaneously by alternating between the processes and executing small portions. In a parallel processing system, multiple CPUs execute multiple processes residing computer memory. To be truly parallel, the processes must all work together to perform some common function.

Object oriented programming utilizes processes in single process mode. Objects are implemented through independent processes, which conceptually execute simultaneously. The processes communicate with one another through message passing. When an object passes a message to another object, it is requesting that object to manipulate its data with its methods. This is illustrated in Figure 13.

To summarize, the execution of an object oriented program involves a set of independent processes working together to perform some function.

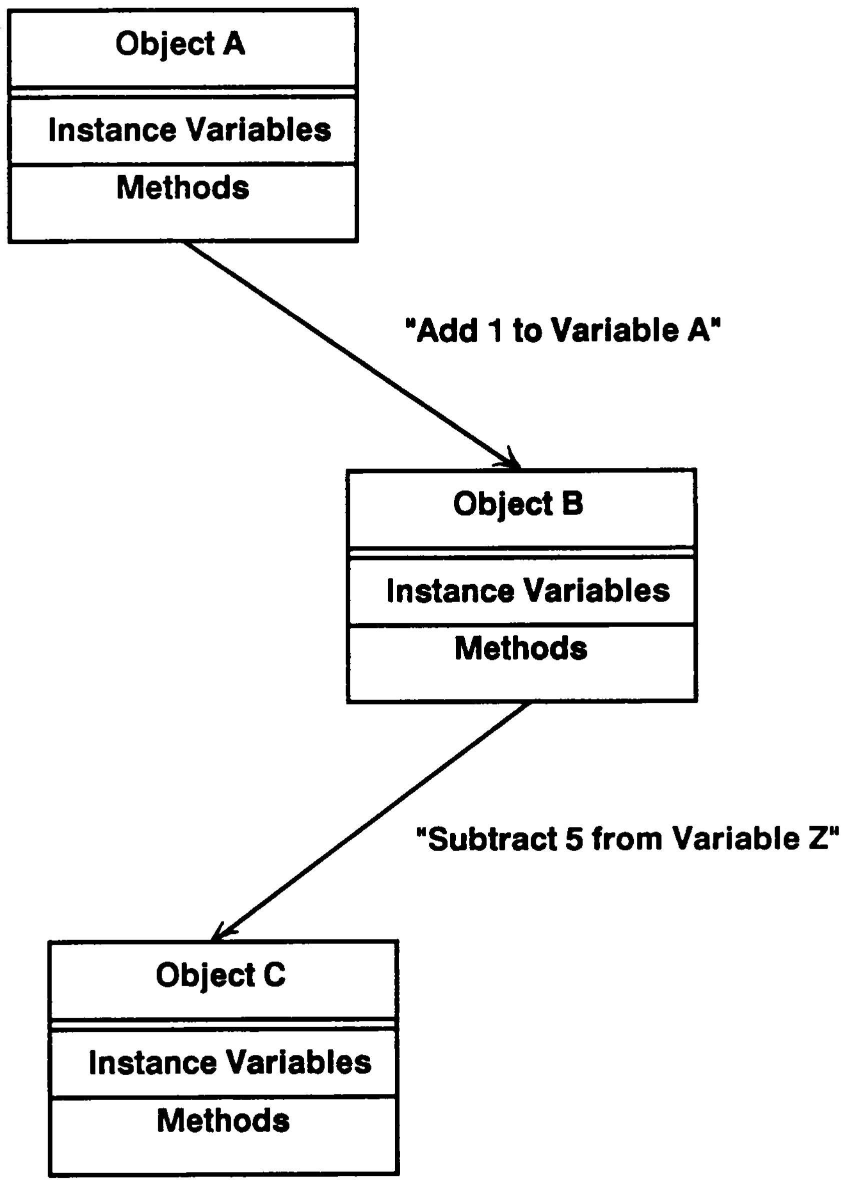


Figure 13 Message Passing Among Objects

2.1.1.4.1 DEVS World View

The major characteristic of DEVS is that it permits the hierarchical, modular construction of discrete event simulation models. The following description refers to Figure 14.

Consider two separate models, A and B. These models are called **atomic models** and each represents some sub-system within a system. Each atomic model can be considered as an independent entity which communicates with its environment through its set of **input/output ports**. During execution, an atomic model monitors its input ports, simulates activities that relate to its sub-system, and provides values to its output ports. By coupling the input ports of B to the output ports of A (and/or possibly vice versa), both models can now work together to simulate their respective sub-systems. This type of coupling is called **internal coupling.**

Now suppose that it is desired to create a new model, AB, from the coupling of models A and B. By coupling the input ports of AB to the input ports of A and B (**external input coupling**) and the output ports of A and B to the output ports of AB (**external output coupling**), a completely new model is created which now represents the combined sub-systems of A and B, and which enjoys the same basic frame-work. Such a model is called a **coupled model**.

This scenario can be repeated with coupled models being created from the combining of other coupled models (see Figure 15). Zeigler calls this process **"hierarchical construction."**

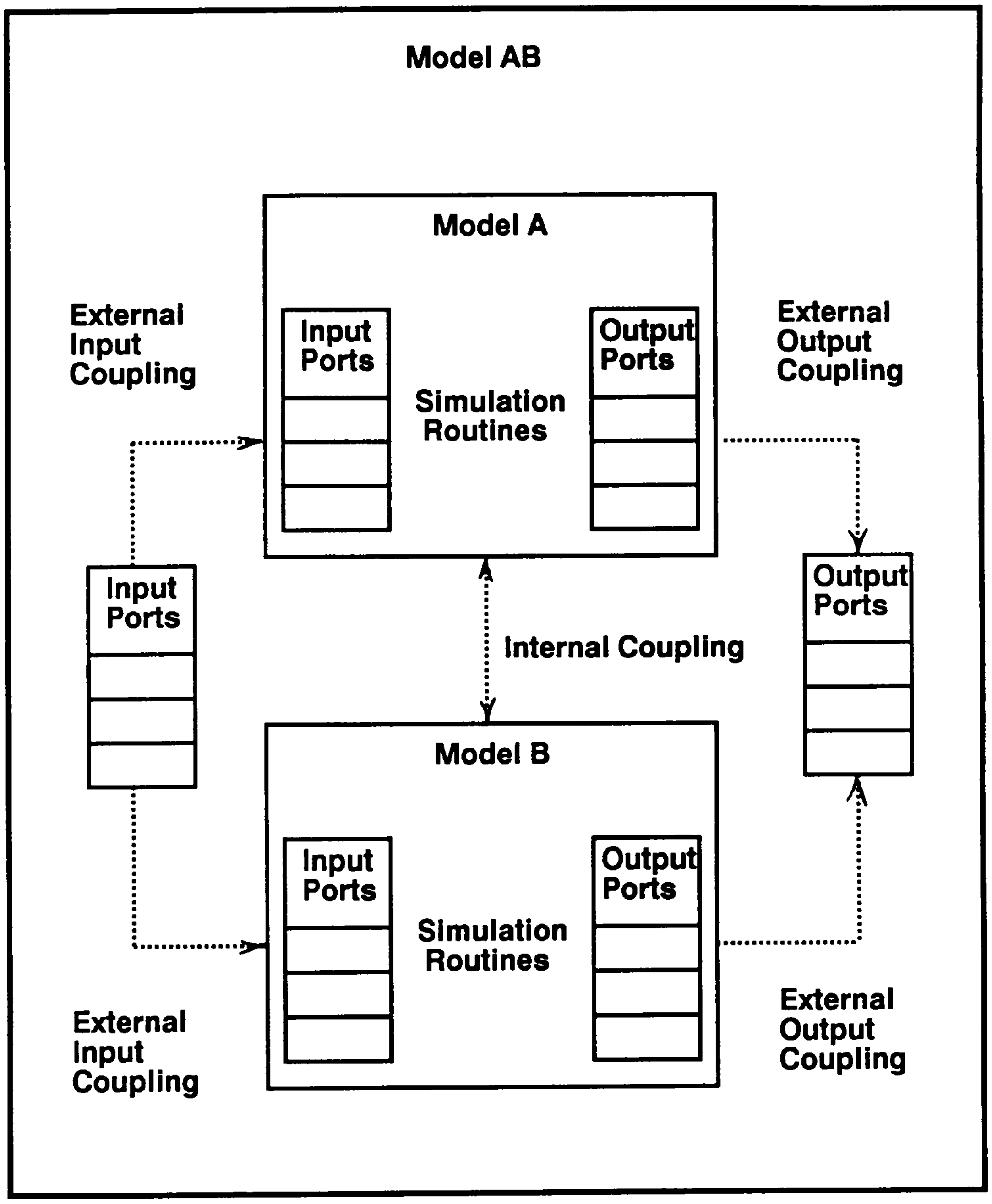


Figure 14 DEVS World View

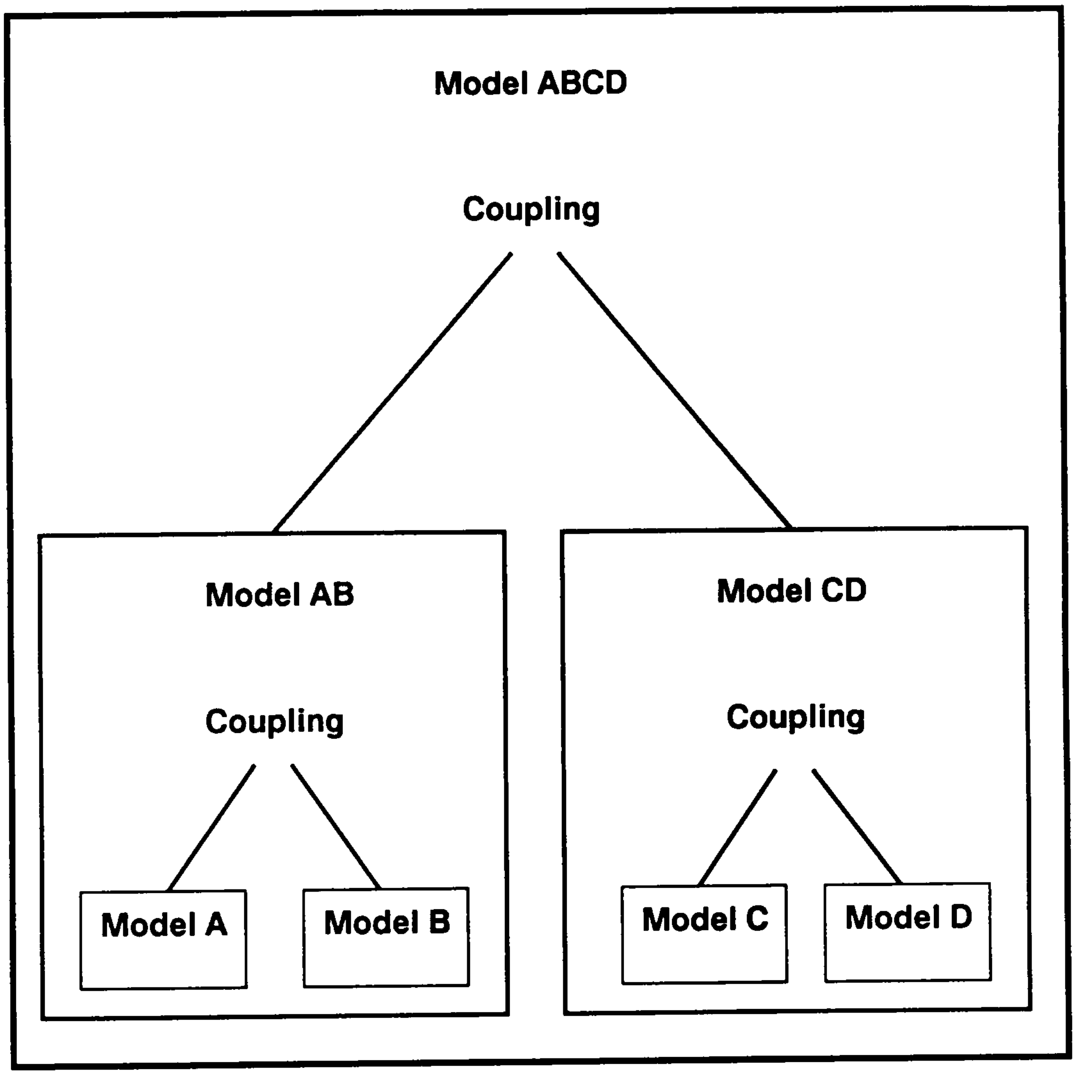


Figure 15 Hierarchical Construction in the DEVS Frame Work

While the DEVS framework can be implemented with a general purpose language, its structure is very conducive for implementation with an object oriented programming language. Atomic models are readily implemented through the use of objects. Coupled models are readily implemented through objects, whose classes are the super-classes of atomic objects. Finally, communication via input/output ports is handled very nicely through message passing.

DEVS has many advantages over the other world views discussed so far. Due to its unique structure, DEVS provides an extremely modular approach to model construction. In addition, due to its distributed nature, DEVS also provides a basis for parallel simulation. Parallel simulation utilizes multiple CPUs executing multiple processes to speed up the execution of a simulation model. This concept has been demonstrated by Concepcion and Schon [11] with their implementation of SAM (Specifier and Analyzer Module), a software environment which aids modelers in the creation of parallel executing, DEVS structured simulation models.

2.2 Continuous Simulation

Continuous simulation models are used to represent the continuously changing behavior of systems over the passage of time. The following describes how they are applied to manufacturing system analysis and discusses their mechanics.

2.2.0 Applications in Manufacturing System Analysis

Continuous simulation models in their basic form are used to analyze a large range of electrical, mechanical, and fluid systems within manufacturing facilities. An example is the analysis of the kinematics and dynamics of industrial machines [10].

In modified form, continuous simulation models are used extensively to analyze the kinematic or dynamic behavior of intelligent industrial machines such as robots and CNC (Computer Numerical Control) machine tools within manufacturing work cells [14,23,26]. These models are used primarily for cycle time analysis and the computation of torques, link velocities, and link accelerations.

In many cases these same models are integrated with geometric modeling techniques to capture geometric information as well. This geometric information is used to layout work cells and to detect collisions between machines and other elements within work cells [18]. Wang [51] has utilized these types of models to compute the forces between a cutter (connected to a kinematic model of an CNC machine tool) and a work piece.

As a result of its extensive use in manufacturing system analysis, many specialized continuous simulation environments have been developed for manufacturing applications. The main applications fall into robotic work cell analysis environments. Examples include IGRIP [34], Robot-Sim [20], RPS [28], ROBOSIM [17], and ROBOTICS [45].

2.2.1 Continuous Simulation Mechanics

Continuous simulation models can be implemented in a number of frameworks. However, a review of the literature suggests that the two most prevalent types can be classified as either **basic** or **modified**. As the name implies, the modified form is a modification of the basic form. These world views are described below.

2.2.1.0 Basic World View

Under the basic world view, system and/or object attributes are modeled as mathematical functions of continuous variables. At least one of these variables is time. The others are either system and/or object attributes. The mathematical relationships are set up as either difference or differential equations [42].

The logic of the basic world view is illustrated in Figure 16. Initialization results in the assignment of values to both system and object attributes as well as initialization of the clock, typically to zero. During the time scan phase, the clock is incremented by a fixed increment.

During the function scan phase, attributes are updated by firing all of the attribute description functions. This firing may require the use of numerical integration techniques for equations in differential form. The time and function

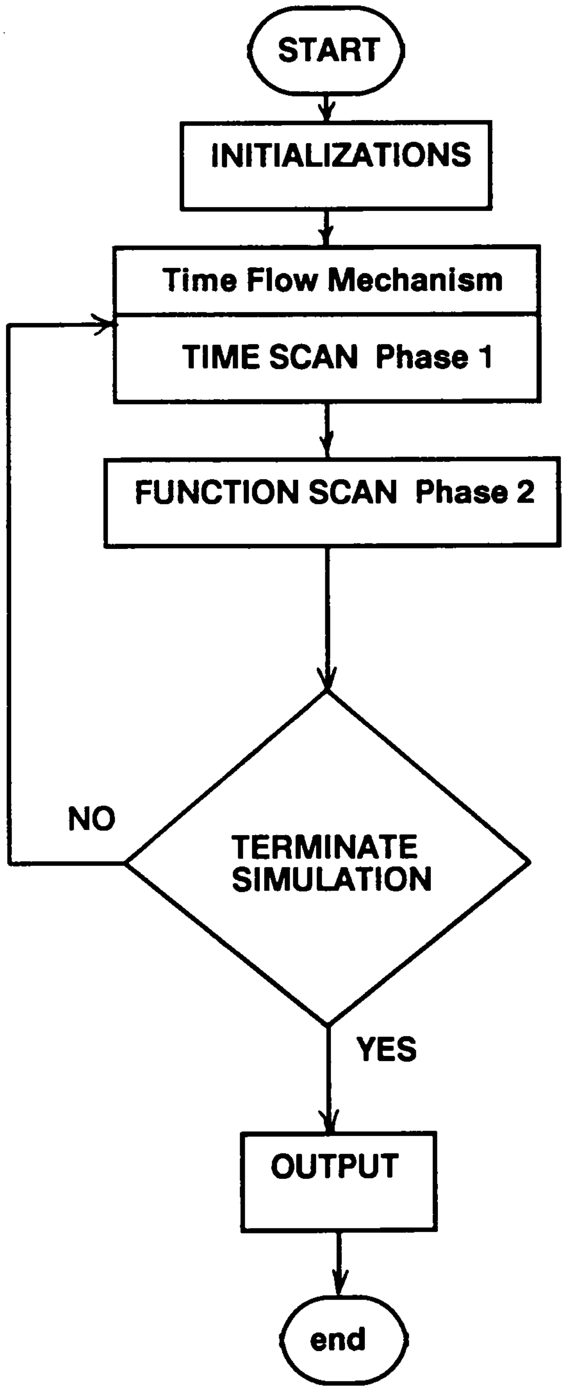


Figure 16 Basic World View

scan phases are repeated until some termination condition is met. This condition is usually the clock exceeding a time limit. Output from the simulation takes place either during the execution of the simulation or upon its termination.

Due to its highly function oriented structure, the basic world view produces highly modular, efficient simulation models. However many manufacturing systems (intelligent machines in particular) exhibit behavior which is both continuous and event oriented. For these systems, the basic world view is inadequate.

2.2.1.1 Modified World View

The modified world view is utilized to represent systems which exhibit both continuous and event oriented behavior. Under the modified world view, a third phase is added to the basic format. This is illustrated in Figure 17. This third phase is an activity scan. During this phase, if the proper conditions are met, an activity fires, resulting in some combination of the following:

1. the update of attributes,
2. the changing of parameters in attribute description functions, and/or
3. the activation or deactivation of attribute description functions.

Like the activity scanning or three phase world views, the activity scan is repeated until all activities fail to fire. The time, function, and activity scan

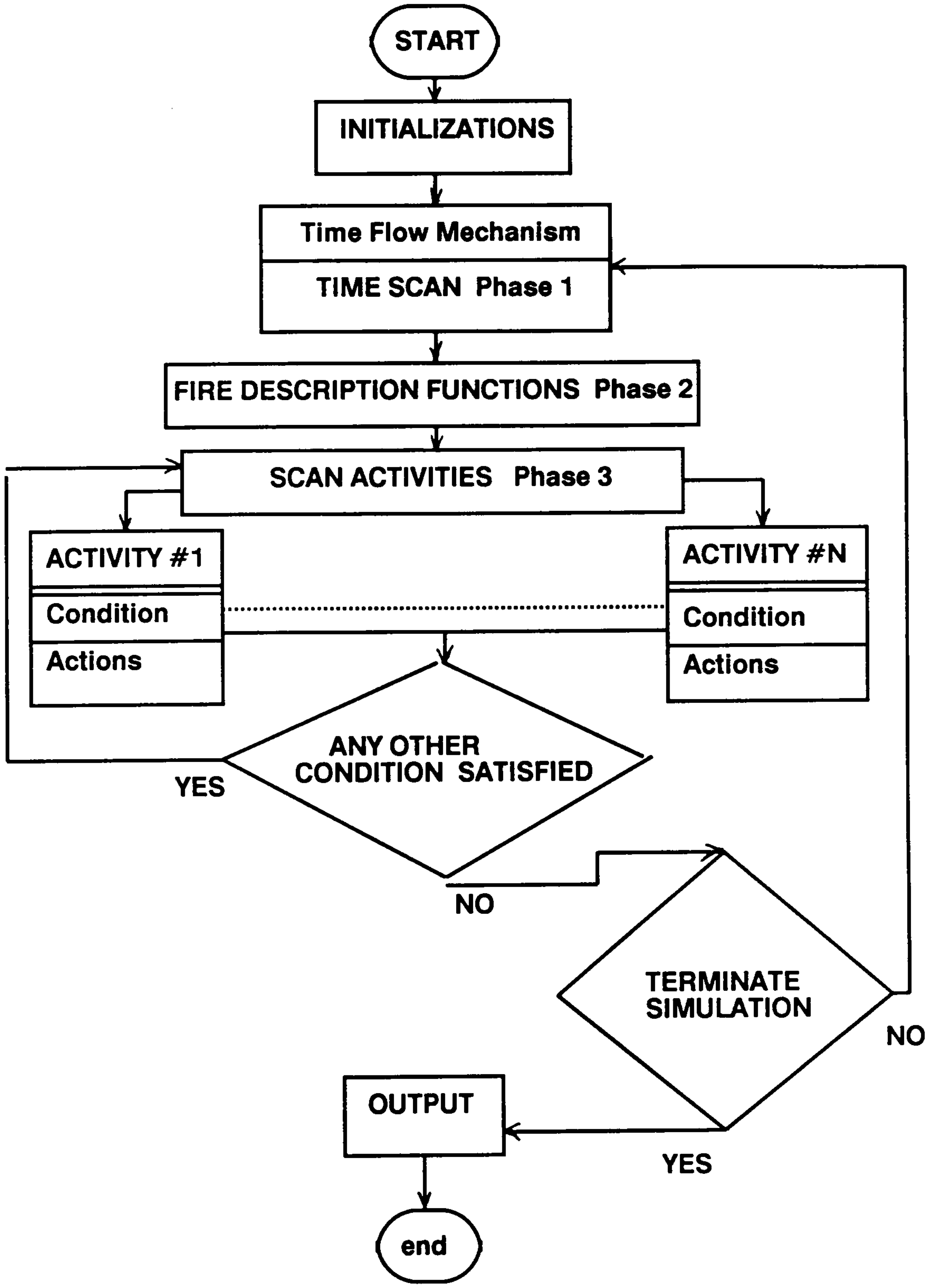


Figure 17 Modified World View

phases are repeated until termination conditions are met. Output is handled in identical fashion as the basic world view.

A simulation of an intelligent machine normally utilizes this world view [23], since the sequential execution of a machine program, which is an event based procedure, results in the alteration of machine trajectory at discrete points in time. In a common implementation, simulated execution of a machine program takes place in the activity scan phase, while the simulated motion of the machine joints and links occurs in the function scan phase.

Models created using the modified world view are very modular and efficient. Note how closely this world view mimics the three phase world view used in discrete event simulation. In fact, the only difference between the two are the actions of the second phases. As a preview to chapter 4, it should be noted that the framework developed in this dissertation is a hybrid of the three phase approach, DEVS, and the modified continuous simulation framework.

The modified world view just presented is not the only method used to combine attribute description functions with activities or events. Pritsker [42] and Pegden [41] both describe a method of integrating the basic world view with the event scheduling world view.

2.3 Geometric Modeling

Geometric models are data objects which describe the geometry of an object in space. Their common use with respect to the simulation of manufacturing systems has been previously described. The remainder of this section will discuss the mechanics behind geometric modeling. The following discussion is taken from Mortenson [36].

The most widely used geometric objects are points, curves, surface patches, and solids. Points are usually represented by coordinates in three dimensional space.

Curves are ordinarily represented parametrically. That is, the coordinates of points that lie on the curve are functions of a separate variable as opposed to an axis coordinate. The number of degrees of freedom needed to define a curve in space is a function of the continuity of the curve.

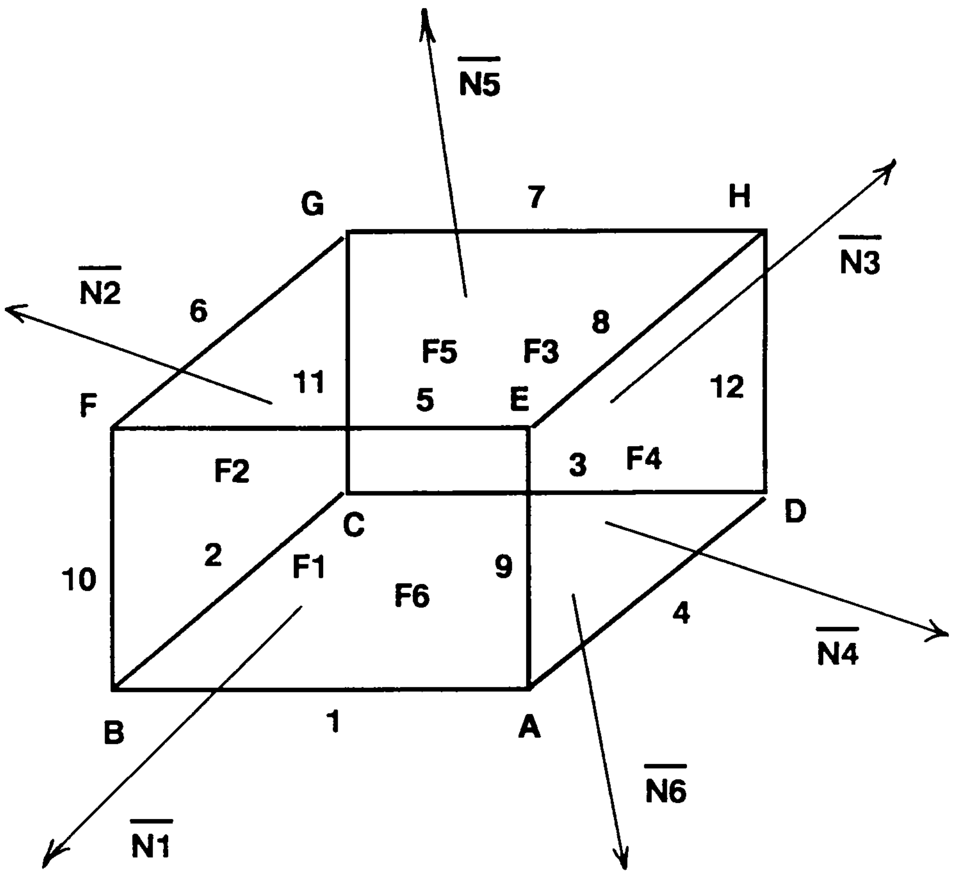
Surface patches are usually parametric with respect to two variables. Points lying on a surface patch are considered to be the intersection of two parametric curves. Like the curve, the number of degrees of freedom required to define a patch is a function of its continuity.

Solids are usually represented in one of two fashions. The first is an analogous mathematical representation as described above. Unfortunately, these representations are limited, since they can only represent solids with no holes, and because they are difficult to alter other than parametrically. Thus, they provide a descriptively weak representation. The more common practice is to use **solids models**.

A solids model represents three dimensional objects by defining all of the topological relationships between the bounding vertices, edges, and surfaces of the object. For example, a cube is defined by its 8 vertices, 12 edges, and 6 faces. This is illustrated in Figure 18. It should be noted that all of the boundaries are described mathematically.

A point in space is considered to be either inside of an object, outside of an object, or on the boundary of an object. To determine the status of a point in space with respect to an object, topological tests are applied to the point and the defining boundaries of the object.

Solids models are typically created and altered through the use of **Boolean operations** applied to their boundaries and the boundaries of other objects. During these operations, boundaries are created, destroyed, and modified. This is illustrated in Figure 19, where object B is subtracted from object A. Boolean operations include the determination of intersections or unions, subtractions, and additions. The determination of intersections between solids models is the method used for collision detection described in section 2.0. Note that the complexity of the boolean operations applied to solids models increases dramatically as the complexity of the representative geometric objects increases.



POINT LIST
A
B
C
D
E
F
G
H

EDGE LIST			
1	$\overline{AB}$	7	$\overline{GH}$
2	$\overline{BC}$	8	$\overline{HE}$
3	$\overline{CD}$	9	$\overline{AE}$
4	$\overline{DA}$	10	$\overline{AB}$
5	$\overline{EF}$	11	$\overline{AB}$
6	$\overline{FG}$	12	$\overline{DH}$

FACE LIST		
F1	-1,9,5,-10	$\overline{N1}$
F2	6,-11,-3,10	$\overline{N2}$
F3	7,-12,-3,11	$\overline{N3}$
F4	-9,-4,12,8	$\overline{N4}$
F5	-1,-4,-3,-2	$\overline{N5}$
F6	1,2,3,4	$\overline{N6}$

Figure 18 Solids Model of a Cube

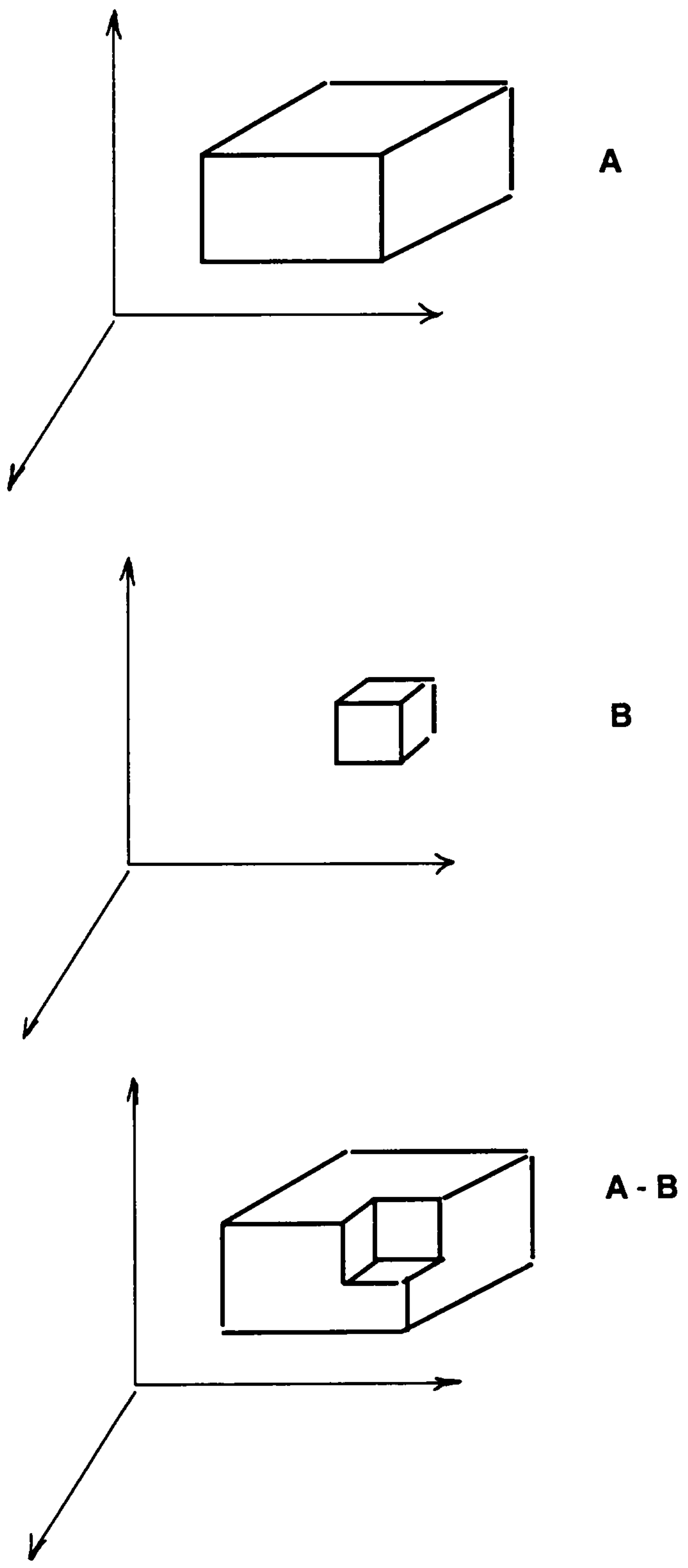


Figure 19 Boolean Subtraction of Solids Models
"

Geometric models are incorporated into simulation models through the treatment of boundary parameters as object attributes. Thus during the execution of a simulation model, the parameters associated with geometric objects change discretely or continuously with the passage of simulated time. Boolean operations on the other hand are not time dependent, but instead are treated as activities. Thus in the kinematic/geometric simulation (utilizing the modified world view discussed in section 2.2.1.1) of a robotic work cell, collision detection is conducted during the activity scan phase.

2.4 Graphical Animation

Graphical animation has gained wide popularity [38] as a complement to the numeric output of discrete event and continuous simulations. This popularity stems from the ability of visual information to enhance the understanding of the dynamics of an executing simulation [21,24,37].

Graphical animation is the result of the dynamic manipulation of graphics models. Graphics models are data objects which contain information needed to render an image on a video display. Today, most graphics modeling is applied to raster technology.

Graphics models are considered to be either two dimensional or three dimensional. Two dimensional models are based on the concept of a two dimensional **world coordinate system** from which there is only a normal **viewing perspective**. This is best illustrated in Figure 20.

Two dimensional objects are created in the world coordinate system and transformed to a **normalized coordinate plane**. In reality, there can be many separate world coordinate systems, all of which are transformed to a normalized plane. The objects on the normalized plane are subsequently transformed to the memory associated with a video display. Two dimensional models contain this coordinate and transformation data.

With respect to manufacturing simulation, two dimensional graphical animation is typically used in combination with discrete event models to represent resources and parts in a queueing network representation. An example of a snap shot of such a representation can be seen in Figure 2.

In the process interaction framework, update of the graphics models is conducted at the end of the process scan phase. In the event scheduling approach, manipulation of the graphics models must occur near the end of event executions. Note that in both cases, real time graphical animation is assumed. In many cases, the output from completed simulations are fed to postprocessors, which then produce a graphical animation.

Three dimensional graphics models are based on a three dimensional world coordinate system and a variable viewing perspective. This is best illustrated in Figure 21. Three dimensional objects are defined in a world coordinate system. Their image is projected along a viewing vector to a normalized viewing plane.

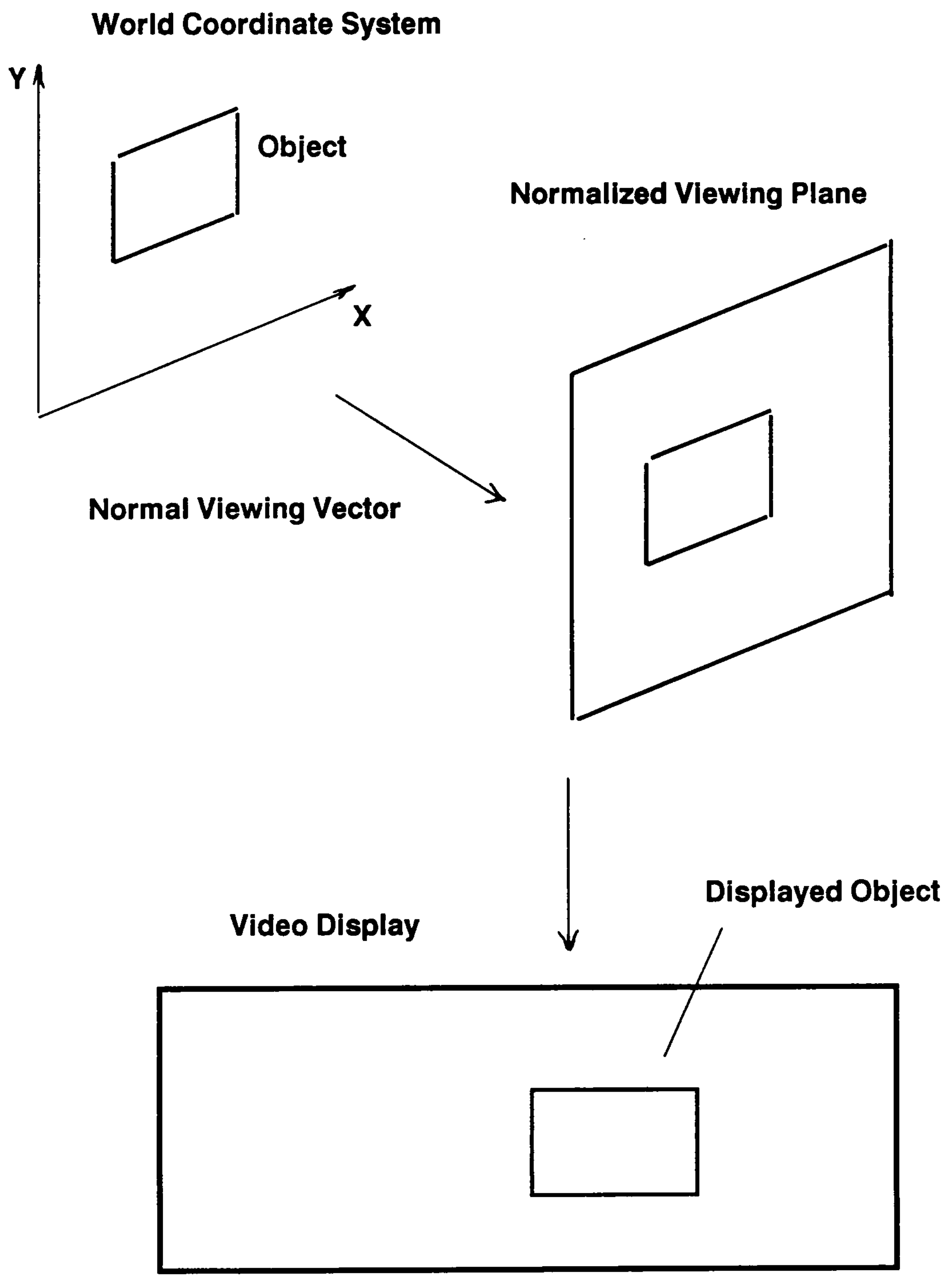


Figure 20 Two Dimensional Graphics

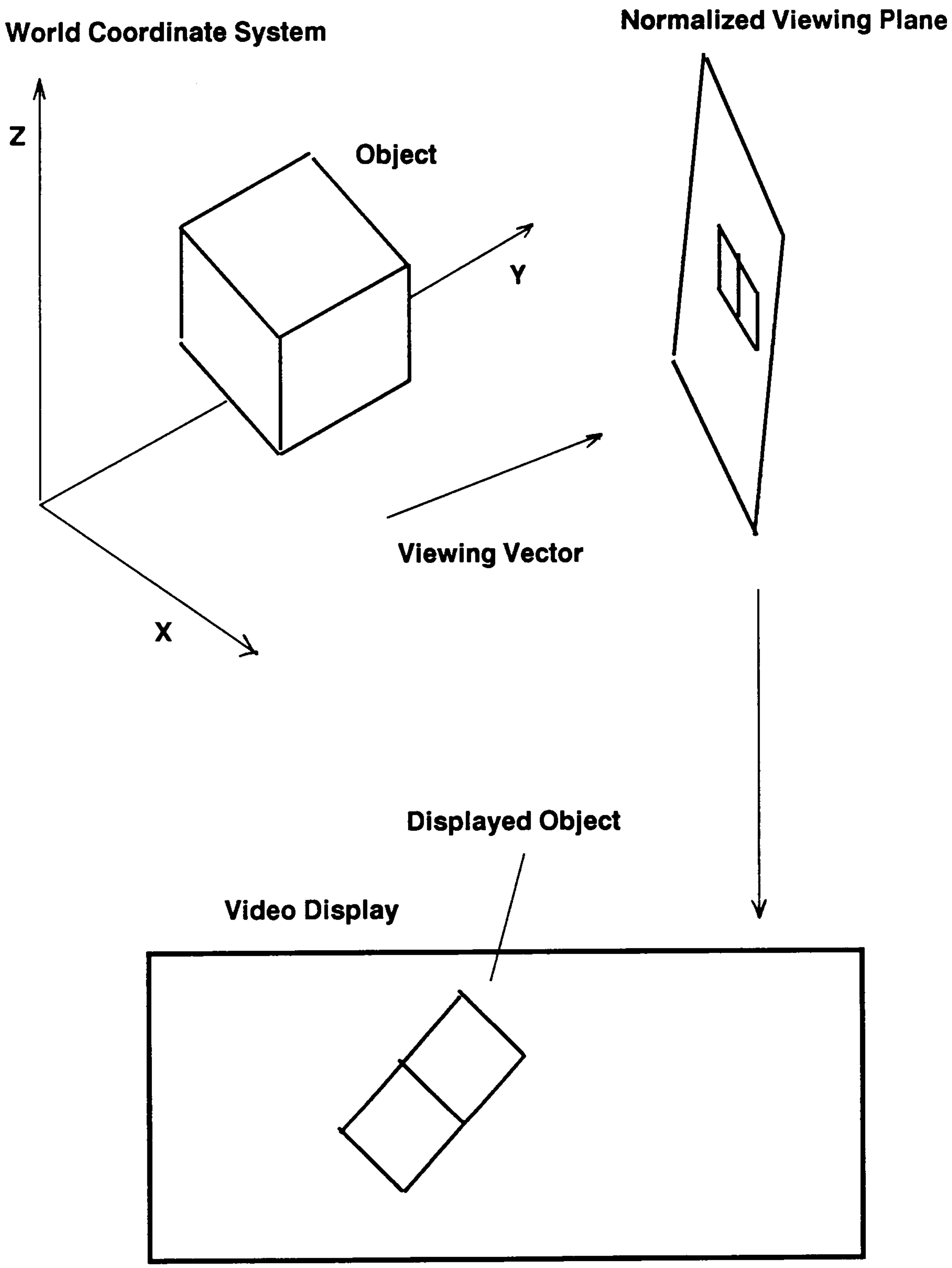


Figure 21 Three Dimensional Graphics

Like two dimensional models, there are usually many world coordinate systems, all of which project on to a normalized viewing plane. Similarly, the projections on the viewing plane are transformed to the video display.

The representation of surfaces of three dimensional objects can be done with either **wire frame** (see Figure 21) or **shaded images**. Shading involves the use of mathematical algorithms to simulate the visual image humans see when light reflects from a surface. Modeling involves computing the reflective intensity of light (from a point source) reflecting from a modeled surface. This reflective intensity is typically a function of the intensity of the light striking the surface as well as the textual characteristics of the surface being displayed.

With respect to manufacturing system simulation, three dimensional graphical animation is commonly used by robotic simulation environments to provide a visual representation of work cell geometry, and to high light collisions between objects [34]. Update of graphics models occurs during the activity scan phase.

2.5 Automatic Computer Model Generation

A review of the literature suggests that with respect to computer modeling and manufacturing system analysis, problems with modularity have been confined to the implementation of discrete event simulation models. As was mentioned in section 2.0.1.4, the vast majority of these models utilize either the process interaction or event scheduling world views, both of which are extremely non-modular.

Historically, the approach taken by researchers to overcome this problem has been the development of automatic model generation software based on either the event scheduling or process interaction world views. This is opposed to the utilization or development of more modular simulation frameworks. The reason for the phenomenon is not apparent.

An automatic computer model generator [7] is "an interactive software tool that translates the logic of a model described in relatively general symbolism into the code of a simulation language, general purpose language, or machine language." Its purpose is to relieve a modeler of much of the effort required to create a computer simulation model, much in the same way that a general purpose simulation language does, but only to a much greater extent.

The historical development of automatic model generation has not been directed at solving the problem of modularity, but rather transferring the burden of model creation from a modeler to a software tool. As a result, model alteration performed by an automatic model generator requires the same amount of model code restructuring (if not more) as a manual approach. The only difference is that with the automatic approach, the modeler is spared the details. Note that in many instances, manual alteration of models can be performed much faster than automatic alteration.

Automatic computer model generators are **domain specific**, which means they are limited to the description of a particular class of systems, such as flexible manufacturing systems, job shops, etc. A model generator can not be used to create models outside of its description domain. Increasing the domain of a model generator requires alteration of the generator itself. The effort required to do this is dependent upon the modularity of the generator code and/or the modularity of the model framework upon which it is based.

A paradigm of an automatic model generator is presented in Figure 22. A generator consists of two modules, a front end dialog module and a code generation module. The function of a dialog module is to acquire and translate a model specification. The method in which it acquires a specification varies. Some accept a textual source program, others allow a modeler to interactively specify a model through the manipulation of graphical icons, and still others allow a modeler to interactively specify a model through natural language constructs. The function of the code generation module is to accept a translated specification and to generate a computer model. Methods of generation vary. Some attempt to link together existing segments of model code while others take a "start from scratch" approach. Information on the mechanics of program generation can be found in Balzer [7].

Examples of manufacturing specific, automatic model generators include those developed by Haddock [25], Medeiros and Sadowski [32], and Ford, Schroer, and Daughtrey [19].

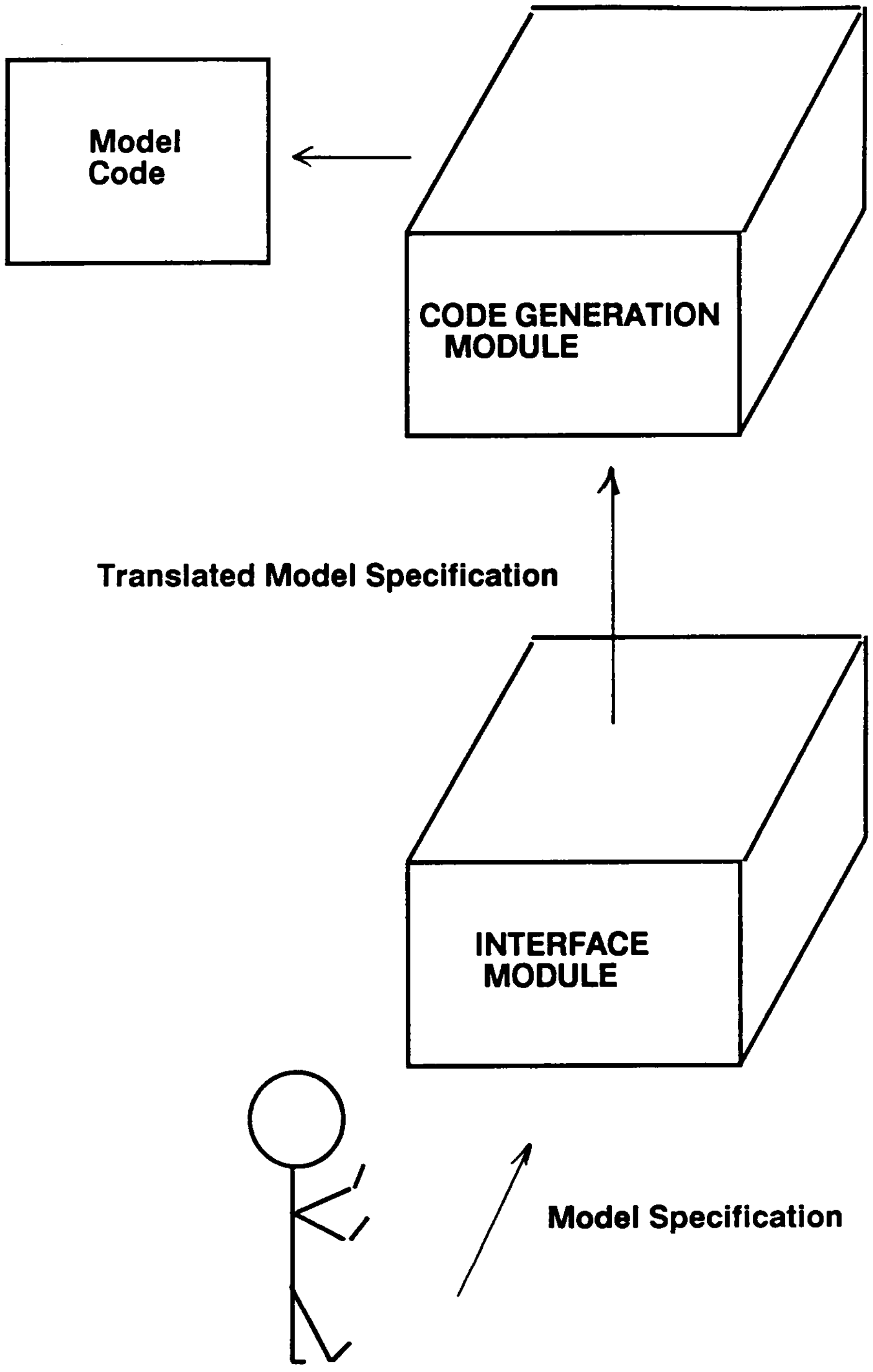


Figure 22 Automatic Model Generator Paradigm

Haddock's system was designed for the representation of flexible manufacturing systems. Its dialog module accepts a textual source program utilizing generator specific symbols. Its code generation module produces a simulation model written in the SIMAN programming language.

Medeiros and Sadowski's system was designed for the representation of simple robotic work cells. Its dialog module allows users to specify a simulation model through graphical icons. Its code generation module creates code by modifying and merging pre-existing textual source programs written in the QGERT programming language.

Ford, Schroer, and Daughtreys' system was designed for flexible manufacturing systems. Its dialog module utilizes an interactive natural language interface. Its code generation model produces a simulation model written in the SIMAN programming language.

2.6 Simulation Frameworks Developed for Multi-level Design

Three simulation frameworks have been reported in the literature which have been described in correspondence with some form of multi-level manufacturing system design process. These frameworks are the ones developed by the United States Air Force under the ICAM project [30,49,509], Antonelli, Volz, and Mudge [2], and Imam, Davis, and Fougere [27].

The U. S. Air Force sponsored the ICAM project in the early 1980s in order to develop manufacturing system design methodologies which would facilitate the rapid and flexible manufacture of aircraft and avionics. The IDEF methodology was born from this project. IDEF consists of three phases, IDEF0, IDEF1, and IDEF2. IDEF0 provides a methodology for the modular, multi-level functional design of manufacturing systems. IDEF1 provides the same methodology for information flow within manufacturing systems. IDEF2, which is a discrete event simulation methodology, was designed for the analysis of IDEF0 and IDEF1 designs.

The intent of IDEF2 was not the development of a general purpose simulation framework. Instead it involved the adaptation of existing discrete event simulation techniques to the IDEF methodology. The discrete event simulation framework chosen was process interaction. To be even more specific, SLAM [42] network constructs were chosen to represent IDEF entities.

Antonelli, Volz, and Mudge developed a discrete event simulation framework capable of supporting a multi-level, functional design methodology of their own creation. Their framework was based on the integration of the event scheduling world view with the ADA programming language. In particular, they were interested in utilizing the object oriented programming characteristics of ADA to support the concept of a distributed simulation, much in the same fashion as Concepcion and Schon [11]. No emphasis was placed on the modular construction of models however, nor the possible extension of the framework to include languages other than ADA or modeling techniques other than event scheduling.

Imam, Davis, Fougere described the need to integrate discrete event simulation for the analysis of initial designs of manufacturing systems and continuous simulation for the analysis of more detailed designs. They discussed their intent to create a software environment which provided either technique on a separate basis. The concept of a generalized simulation framework was not discussed, nor the need for modular model construction.

2.6 Summary

The computer modeling techniques commonly applied to manufacturing system analysis are quite varied in terms of applications and mechanics. To fully support the modular design process described in section 1.4, a simulation framework must encompass all of these techniques as well as support the modular construction of simulation models. Historically, automatic model generation has been used in part to overcome the inadequacies of conventional simulation frameworks. In addition, limited efforts have been made to adapt discrete event simulation techniques to multi-level functional designs of manufacturing systems. However in general, no simulation frameworks have been developed which can support the modular construction of multi-level simulation models of manufacturing systems. The intent of this research is to develop such a framework. The next chapter discusses the methodology used to develop this framework.

3 METHODOLOGY

3.0 Introduction

Chapter 1 discussed the need for the development of a simulation framework which can support the modular, multi-level design of manufacturing systems. In general, this framework needs to support the modular construction of uni-level and multi-level simulation models. This chapter discusses the methodology used to create this framework.

The topics discussed are: the development of a new simulation framework as a solution to the research problem as opposed to automatic model generation using conventional frameworks, the required characteristics of the framework, the test bed approach used to develop the framework, the hardware, software, and design requirements, and finally the procedure used to demonstrate the framework.

3.1 Framework Versus Automatic Model Generation

Historically automatic model generation has been used in part to overcome the lack of modularity of conventional simulation frameworks, and thus presents itself as a logical solution alternative to the development of a new simulation framework. However the latter approach has been chosen because of its generality and its ability to serve as a basis for efficient, expandable model generators in the future.

With respect to generality and their common use in manufacturing system simulation, automatic model generators are limited to the description and specification of a limited set of manufacturing systems, and create output which is limited to a set of target languages. Research into the use of automatic model generation (utilizing conventional simulation frameworks) as a solution to the stated problem would entail the development of a prototype generator for a specific system domain and the utilization of a specific target language, or the development of a generalized approach for model generation.

The former approach is considered of limited value, since there is no guarantee that the mechanics developed would be applicable to other system domains and/or target languages. The latter approach, while being general in nature, is considered to be extremely difficult due to the myriad of manufacturing system configurations and available target languages.

The development of a new simulation framework on the other hand has the advantage that it is neither domain specific, target language specific, or machine specific. If properly designed, the framework should be applicable to the simulation of any manufacturing system at any level of detail on any machine.

In many cases model alteration is faster through the manual alteration of manually created models than alteration via automatic model generation. This is due to the long execution time of many generators. For any given simulation, the execution time of a generator is directly related to its **efficiency**.

Generator efficiency is defined as the average number of computer instructions that are executed for the generation of a simulation model of a standard system. Generator efficiency is directly related to at least two properties of its target language. They are the applicability of its target language with respect to the modeling task, and the modularity of the language.

Target languages (based on conventional frameworks) that are limited to particular techniques or have limited sets of modeling constructs may not be applicable for the modeling of particular systems, even when the systems are within the domain of the generator. As a result, the task of model generation becomes more complex, difficult, and demanding of computer time relative to the generation of models in more applicable languages.

Likewise, target languages that are not modular make the task of model generation more difficult since the ability of a generator to utilize exi$\overset{s}{\wedge}$ting sub-models is either limited or non-existent. In the former case, greater generator complexity is needed to resolve dependencies between sub-models and to glue sub-models together. In the latter case, generation involves the creation of a totally new model. In both cases, greater amounts of execution time are needed relative to the generation of models in more modular languages.

The development of a new simulation framework has the advantage that it not only provides a solution to the research problem, but due to its generality and modularity, should also serve to improve the efficiency of model generators in the future.

The **expandability** of the domain of an automatic model generator is dependent upon the modularity of its construction as well as the generality and the modularity of its target language(s). As was just discussed, the development of a new simulation framework promises to improve the generality and modularity of future target languages. Thus it is assumed that it will also promote the expandability of future model generators.

In conclusion, the development of a new simulation framework is considered to be a superior solution to the research problem than the development of an automatic model generation methodology based on conventional simulation frameworks. This is due to the more general nature of the former approach and the fact that it can serve as a basis for efficient, expandable model generators in the future.

3.2 Required Characteristics of the Simulation Framework

To support the modular construction of uni-level and multi-level simulation models, the simulation framework developed in this research must exhibit a large degree of **generality** and **modularity**. In an effort to make these characteristics more tangible, this section discusses them in detail.

3.2.0 Generality

In terms of generality, the simulation framework must be able to encompass all of the computer modeling techniques applied to manufacturing system simulation. Examples include discrete event simulation, continuous simulation, and geometric modeling. A review of the literature suggests that this can be accomplished if the framework is based on the execution of activity scans, event functions, and attribute description functions, as well as the incorporation both fixed increment and variable increment time flow mechanisms.

In addition, it is desired that the framework be applicable to multi-process simulation as well as single process simulation. While multi-process capability does not expand the domain of the framework, it is important none-the-less. The execution time required for large scale simulations is enormous [55]. Parallel simulation shows great promise for reducing the execution time of large scale models. Since many manufacturing system simulations can easily be considered large scale, and because development of parallel processing technology is progressing at a rapid rate [1], it is believed that the new framework should take advantage of parallelism within manufacturing system simulation models.

Based on these requirements, it is believed that the simulation framework should incorporate features of the three phase (discrete event simulation) and modified (continuous simulation) world views, since together they meet all but the multi-processing requirements discussed so far. Note that both world views are essentially activity based, which plays an important role in the achievement of framework modularity.

3.2.1 Modularity

With respect to modularity, the simulation framework should tend to maximize code reusability so that a computer model can be created from a composition of existing sub-models, and that its alteration is limited to the replacement of a set of these sub-models. In addition, the framework should also permit the stand-alone execution and validation of sub-models. The importance of this concept was discussed in section 1.2. To obtain this level of modularity, the simulation framework must not only be modular with respect to its mechanics, but it must also take advantage of any potential modularity in the conceptualization of a manufacturing system model.

With respect to simulation mechanics, modularity is easily obtained from the hybridization of the three phase and modified world views, since both are individually modular. Conquering the other modularity issue is not quite as apparent, especially utilizing the model conceptualizations offered by conventional simulation frameworks. However the DEVS framework utilizes a model conceptualization which offers greater insight into solving this problem. This conceptualization involves atomic models and internal coupling.

In DEVS, an atomic model is considered a separate entity which simulates the internal interactions of an object within a system. Internal interactions may be conditional in the sense that they are triggered by changes in the model environment, or time based in that they are triggered by the passage of time. An atomic model senses change in its environment through communication via input/output ports.

Separately, two atomic models can simultaneously simulate two independent objects. However they would fail to properly simulate a system if the two objects were dependent upon one another, because interactions between the objects are neglected. To overcome this, internal coupling is applied to the two models. Coupling involves the transfer of values from the output ports of atomic models to the input ports of other atomic models. To do this, the I/O ports of various atomic models must be mapped together. In this fashion, an entire system can be simulated by linking the inputs and outputs of individual sub-system models.

If this coupling concept were to be expanded to include not only the mapping of ports, but also the monitoring of the complete states of atomic models and the simulation of the interactions between atomic models, a conceptualization very analogous to the normal interactions within a manufacturing facility is created. Neglecting the concept of input/output ports temporarily, resources within manufacturing centers can be considered as separate entities which can operate not only in stand alone fashion, but through interactions, whether logical or physical, in a dependent fashion.

The term "resource" applies to any element, at any level of abstraction within a manufacturing facility. Examples of resources could include an entire plant, a work cell, a robot, a motor, even a resistor. It does not matter, because all are sub-systems which can act alone and in combination with others.

Note the terms **internal interactions** and **external interactions**. Internal interactions represent the interactions that take place within resources while external interactions represent the interactions between resources. This concept is critical to the exploitation of new avenues of modularity within manufacturing system simulation models and is the corner stone upon which the new simulation framework is based.

An important side issue is the concept of input/output ports. While developed by Zeigler to take advantage of message passing in the execution of object oriented simulations, it is also very relevant to the operation of manufacturing facilities. During normal activity, resources communicate with one another through channels of communication in order to maintain synchronization. For a technician, these channels may be in the form of a standard operations sheet handed down from a foreman or the appearance of signal kanban. For a robot, these channels may be in the form of binary or serial input/output ports, which allow it to interlock with cell controllers, programmable controllers, or work cell sensors.

Regardless of form, communication through channels is a large facet of manufacturing system behavior. As a result of its applicability, the concept of input/output ports is also included in the simulation framework.

As a final issue, the parallel processing requirement is addressed once again. To create a truly parallel simulation model, individual processes must work together to simulate an entire system. As was demonstrated by Concepcion and Schon [11], the atomic model concept in conjunction with message passing provides a basis for parallel simulation. As will be discussed in chapter 4, the new simulation framework uses the atomic model concept in combination with **shared memory** to provide a different framework for parallel simulation.

3.3 Test Bed Methodology

A test bed methodology was used in the development of the simulation framework. This methodology involved the application and testing of framework concepts to the development of simulation models of a hypothetical manufacturing system. In general, the approach taken was to first develop three separate uni-level simulation models of the manufacturing system followed by the development of an assortment of multi-level simulation models.

The simulation framework was developed using single process techniques in order to keep the research tractable. However as will be discussed in chapter 4, the framework is generalized to both single process and multi-process configurations. The implementation of the framework in a multi-process configuration will be conducted as future research.

The remainder of this section describes the hypothetical manufacturing system as well as the uni-level and multi-level simulation models used in this research.

3.3.0 Hypothetical Manufacturing System

The hypothetical manufacturing system is shown in Figure 23 and consists of two work cells separated by three pallets. Each work cell consists of a three axis Cartesian robot and a three axis vertical milling machine. The role of the work cells is to transform standard raw materials into two end products. Note that three dimensional sketches of the manufacturing system components and parts are provided in Appendix A.

Each pallet in the system has a capacity of two parts. Limit switches are placed on each pallet to detect the presence of parts and to provide signals to the robots.

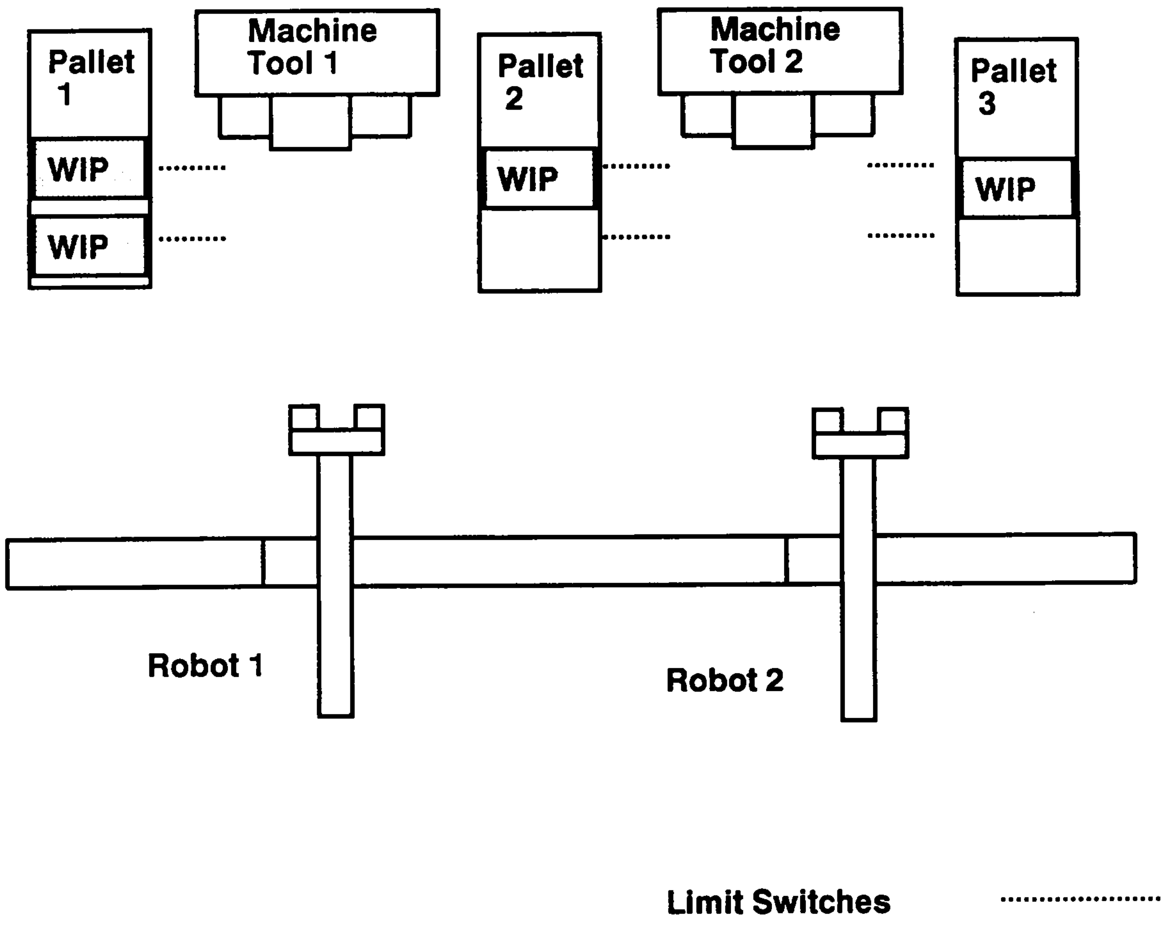


Figure 23 Layout of Hypothetical Manufacturing System

Raw materials enter the system by appearing at pallet 1. These entry materials are considered to be type "0" parts. The robot in work cell 1 detects the presence of the parts via the limit switches, picks up the first part, loads it onto the machine tool, sends a signal to the machine tool to begin execution, waits for a signal from the machine tool indicating task completion, unloads the part from the machine tool, and transfers the part to pallet 2. It then proceeds to go through an identical sequence with the remaining part at pallet 1.

The robot and machine tool in work cell 2 begin a similar operational sequence when the robot senses that two parts are present at pallet 2. The finished parts are placed on pallet 3, where they are removed from the system in pairs. To avoid the possibility of a collision between the robots at pallet 2, robot 1 only initiates the loading of parts on the pallet when it is completely empty, while robot 2 only initiates the unloading process when it senses that the pallet is completely full.

The first part removed from pallet 1 is processed on machine tool 1 into a type "1" part. This part is eventually transferred to machine tool 2, where it is processed into a type "3" part. The second part removed from pallet 1 is transformed by machine tool 1 into a type "2" part and then transformed by machine tool 2 into a type "4" part. This cycle is repeated indefinitely.

3.3.1 Uni-level Simulation Models

Three uni-level simulation models of the hypothetical manufacturing system were chosen as the initial test bed for the development of the framework.

These three levels were chosen to represent the computer modeling techniques commonly used in manufacturing system analysis.

The **level 1** representation, which is shown symbolically in Figure 24, utilizes discrete event simulation and the queuing network analogy. Here the work cells and limit switches are represented as servers, the pallets as storage, and the parts as customers. A partial list of attributes for the simulation entities is provided in TABLE 3.

The **level 2** representation of the hypothetical manufacturing system utilizes the same discrete event/queuing network analogy as the level 1 representation. The only difference is that the work cells have been divided into their respective components. This is shown in Figure 25.

The **level 3** representation, utilizes continuous/geometric modeling techniques to represent the kinematic/geometric behavior of the robots, machine tools, pallets, and parts. The sensors are modeled in the same fashion as levels 1 and 2. A partial list of attributes for the simulation entities is presented in TABLE 4.

Solids models are used to represent the geometry of system components. Boolean intersection, subtraction, and addition are used to model the cutting of parts by the machine tools. A simpler process, collision detection, was not implemented due to its potential to slow down the execution of the simulation model.

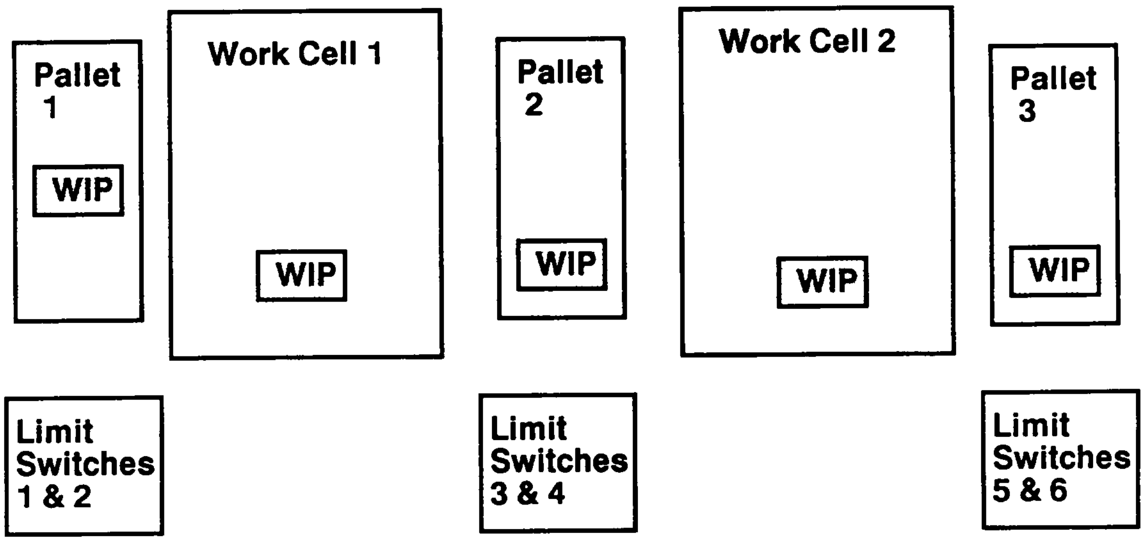


Figure 24 Level 1 Uni-Level Simulation Model

TABLE 3 Attributes for Levels 1 and 2 Uni-Level Simulation Models

Entities	Attributes
Work Cell	Busy/Idle Status
Robot	Instructions
	Instruction Pointer
Machine Tool	Symbolic Location
	Current Number of WIP in System
Pallet	Capacity
	Current Number of WIP on Pallet
	Symbolic Location
Part (WIP)	Current Location
	Current Part Type
Limit Switch	WIP Detection Status
	Symbolic Location

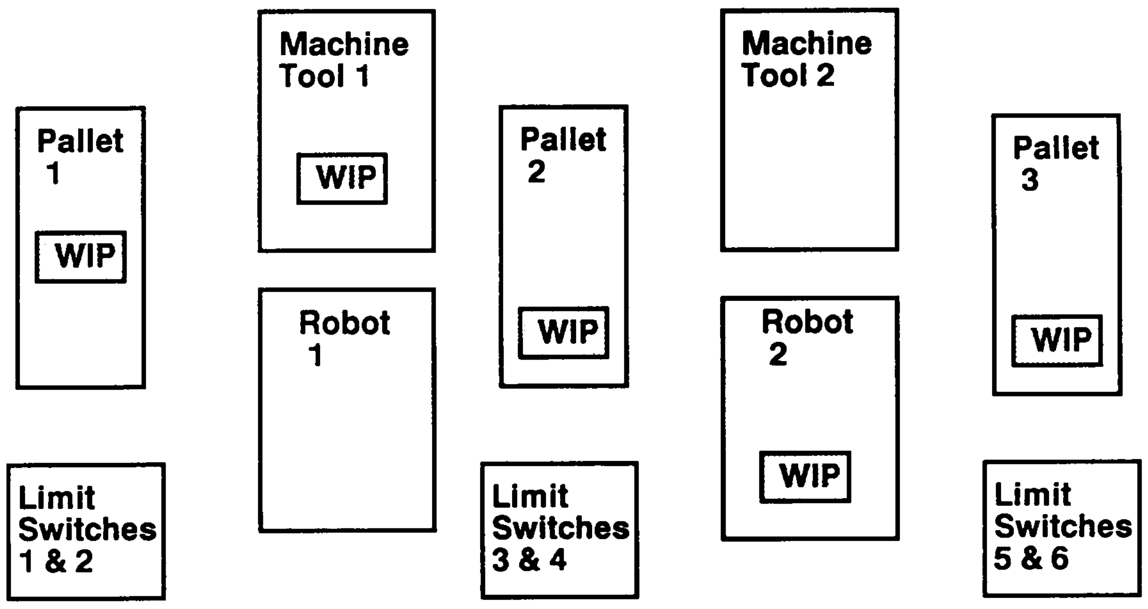


Figure 25 Level 2 Uni-Level Simulation Model

TABLE 4 Attributes for Level 3 Uni-Level Simulation Model

Entities	Attributes
Robot	Position of Base Link
	Position and Velocity of Joints
Machine Tool	Position and Velocity of Links
	Solids Models of Links
	Instructions
	Instruction Pointer
Pallet	Current Number of WIP on Pallet
	Position and Velocity
	Solids Model of Pallet
Part (WIP)	Position and Velocity
	Solids Model of Part
Limit Switch	WIP Detection Status
	Symbolic Location

Note that graphics models were not included in the research due to the lack of graphics libraries. The construction of graphics libraries would have been possible, but it would have not contributed much to the research, while expanding the research effort considerably.

In general, the development process during this stage consisted of iteratively changing and experimenting with the uni-level simulation models in order to:

1. determine the validity of the conceptual framework, and
2. improve the modularity and efficiency of the framework mechanics.

3.3.2. Multi-level Simulation Models

Upon evaluation of the framework with respect to the uni-level simulation models, further evaluation was conducted on 22 different multi-level simulation models. These models represented all of the possible combinations of the work cell, robot, and machine tool models established in the uni-level simulations. A complete component summary for each of these models is presented in Appendix D.

Framework evaluation was conducted in a fashion similar to the previous stage. As it turned out, the complete development of the framework required three iterations of the combined stages.

3.4 Hardware, Software, and Design Requirements

To conduct the research, a number of prerequisites had to be satisfied. They were the selection of computer resources, the selection of computer software, the creation of a library of geometric modeling routines, and the design of the hypothetical manufacturing system. This section discusses these topics in detail.

3.4.0 Hardware Selection

The computer selected to conduct this research was an AT&T 7300 UNIX PC. This computer was used because it was the only type of machine in the manufacturing system laboratories (outside of the network server) which ran the multi-tasking operating system, UNIX. At the time of selection this was important, because the original intent of this research was to develop the framework using multiple processes. While this expectation was not met, the UNIX PC still served its purpose well.

3.4.1 Software Selection

The software used to conduct this research was a C language compiler. This software was selected because it was the only compiler available on the UNIX

PC. Fortunately, C is a powerful, pointer based, general purpose programming language which permitted the rapid construction of simulation code.

3.4.2 Geometric Modeling Routines

The level 3 uni-level simulation model of the hypothetical manufacturing system requires the use of solids models and Boolean routines. No such routines were available for the UNIX PC in terms of source code. Thus it was necessary to develop solids model data structures and a library of solids modeling routines.

The routines and data structures created revolved around the modeling and manipulation of geometric objects that are the Boolean composition of cubes. The solids model data structures are based on both constructive geometry and boundary representation schemes [35]. The Boolean routines are limited to the intersection, subtraction, and addition of cubic objects.

As a result of the limited capability of the geometric routines, the geometry of all components within the hypothetical manufacturing system are based on simple cubic objects, including cutting tools. In all, the development of these routines required two months of work, and resulted in approximately 8000 lines of C code.

3.4.3 Design of the Hypothetical Manufacturing System

The design of the hypothetical manufacturing system was a prerequisite for the development of the uni-level simulation models. The design attempted to incorporate machines and sensors that are commonly found in automated, discrete parts manufacturing systems as well as the manufacturing system laboratories, namely robots, machine tools, and limit switches. The design was limited to the use of intelligent machines since there was no reason to believe that the use of non-intelligent machines would have any bearing on the research. The design did not incorporate human technicians due to the complexity of modeling humans at the kinematic/geometric level.

In general, this effort required:

1. the design of the geometry and kinematics of the robots, machine tools, pallets, and WIP,
2. the geometric layout of the work cells,
3. the process planning and sequencing of WIP,
4. the design of the overall control and synchronization strategy for the work cells,
5. the creation of simulated controllers for the robots, and machine tools, and
6. the creation of instructions for the robots and machine tools.

3.5 Simulation Framework Demonstration

Demonstration of the framework is based on the execution of the test bed uni-level and multi-level simulation models through the use of an interactive model demonstration environment. The goals of the demonstration are to illustrate:

1. the concept of multi-level simulation,
2. the concept of internal and external interactions, and
3. the modular construction of uni-level and multi-level simulation models.

The environment allows a user to select a model through the use of a menu. Upon selection, the environment graphically presents icons representing the sub-models upon which the model is based. By examining the structures of a number of different models, the user should get a strong idea of the modularity of the framework.

Once a user has viewed the structure of a model, the environment executes the model, allowing the user to control the simulation by setting time limits and reading state variables. To facilitate the user's understanding of the executing model, a limited form of graphical animation is incorporated. Through this method, the user should gain confidence in the validity of the model as well as a further understanding of multi-level simulation and internal and external interactions.

3.6 Summary

The methodology used for the development of the simulation framework has been discussed. Topics that have been covered include:

1. the benefits of developing a new simulation framework as opposed to the development of a generalized methodology for automatic model generation utilizing conventional frameworks,
2. requirements of framework generality and modularity,
3. the need for exploitation of internal and external interactions,
4. the prerequisites of computer selection, software selection, geometric modeling routine library creation, and hypothetical system design, and
5. the development of a software environment for the demonstration of the simulation framework.

The next chapter provides a detailed description of the simulation framework developed in this research.

4 SIMULATION FRAME WORK

4.0 Introduction

The simulation frame work developed in this dissertation is entitled, "**GIBSS**," for **Generalized Interaction Based Simulation Specification**. GIBSS is generalized with respect to manufacturing system simulation due to its utilization of activity scans, event functions, attribute description functions, and both fixed and variable increment time flow mechanisms. GIBSS provides for the modular construction of uni-level and multi-level simulation models through its specification of how these models can be divided into sub-models, each of which can be created, executed and validated independently, and then linked together to form composite models. The use of independent sub-models also permits GIBSS to exploit parallelism within manufacturing system simulation models. This chapter discusses GIBSS, its mechanics, and how it is implemented in single process and multi-process configurations.

4.1 GIBSS Fundamentals

Under GIBSS, a simulation model of a manufacturing system consists of three types of sub-models: **passive sub-models, internal interaction sub-models,** and **external interaction sub-models.** Each type of sub-model has a role in the complete representation of a manufacturing system as well as a unique set of properties.

Passive sub-models are data structures which represent elements within a manufacturing system, which within the scope of a simulation, do not exhibit behavior that is describable through the use of activities, events, or attribute description functions. For example, consider the kinematic/geometric simulation of the hypothetical manufacturing system discussed in chapter 3. A part flowing through the system could be represented with a passive sub-model because its kinematic/geometric behavior is strictly dependent upon the actions of robots and machine tools.

Passive sub-models have two unique properties: passiveness and a dynamic attribute structure. Passiveness relates to the inability of the sub-models to alter their own attributes or the attributes of other sub-models during simulation execution. A dynamic attribute structure permits passive sub-models to have their attribute sets (thus their descriptive power) expanded or contracted by other sub-models during simulation execution.

Passive sub-models are further classified as **dynamic** or **static.** Dynamic passive sub-models represent entities such as work-in-progress which exist temporarily in a manufacturing system during a time of study. Static passive sub-models represent entities which remain within a manufacturing system during a time of study. Examples include platforms, tools, racks, and tables.

Internal interaction sub-models are objects (refer to the object oriented programming discussion in section 2.1.1.4.0) which represent the interactions within elements of a manufacturing system, which within the scope of a simulation, exhibit behavior that is describable through the use of events, activities, or attribute description functions. In the example simulation, the robots and machine tools could be modeled with internal interaction sub-models because their representation requires the scanning of activities and the firing of attribute description functions.

The unique property of internal interaction sub-models is their ability to change their own attributes, but not the attributes of others. Thus during the execution of the example simulation, an internal interaction sub-model representing robot 1 simulates the joint and link movements of the robot through the passage of time, but is unable to effect the joint and link attributes of the internal interaction sub-models representing robot 2 and the machine tools or the passive sub-models representing the parts.

The decision whether to represent an element of a manufacturing system with an internal interaction sub-model versus a passive sub-model is dependent upon the scope of the simulation. For example, if the example simulation were to include the thermal behavior of all system entities, then the parts would have to be represented with internal interaction sub-models rather than passive sub-models since attribute description functions are required to represent this behavior.

External interaction sub-models are objects which represent the interactions between elements within a manufacturing system. These interactions must exhibit behavior which is describable through the use of events, activities, or attribute description functions. The unique property of external interaction sub-models is their ability to change not only their own attributes during simulation execution but also the attributes of internal interaction sub-models and passive sub-models. In addition, they have the ability to create or destroy passive sub-models and internal interaction sub-models.

External interaction sub-models are the cornerstone of GIBSS. Their use is paramount to the modular construction of uni-level and multi-level simulation models. Why this is so is demonstrated by both analogy and example.

When using GIBSS, the construction of a manufacturing system simulation model is like the construction a model car. Internal interaction sub-models and passive sub-models are like pre-made parts. In the case of a uni-level simulation model, the parts come from one model kit and naturally fit together. Construction of the model involves the application of glue on mating part surfaces and the assembly of the parts. External interaction sub-models are the glue that binds internal interaction and passive sub-models.

In the case of a multi-level simulation model, internal interaction and passive sub-models are like leftover parts from various kits. Construction of a model car from these parts involves alteration of the mating surfaces (but not the parts in general) as well as the application of glue. In the case of multi-level simulation models, external interaction sub-models serve as both surface alteration and glue.

As an example, consider the simulation of a work cell which consists of a robot and a machine tool. During execution, the robot loads a part onto the machine tool, waits for the machine tool to finish processing, and then unloads the part. A uni-level, discrete event representation of this system is presented in Figure 26. For reference, this simulation is classified as level 1. The robot and machine tool are represented with internal interaction sub-models. These sub-models each have one attribute, their busy/idle status. The robot sub-model simulates four activities:

1. the release of the part to the machine tool,
2. the transmission of a "start cycle" signal to the machine tool,
3. the wait for a "cycle finished" signal from the machine tool, and
4. the removal of the part from the machine tool.

The machine tool sub-model simulates three activities:

1. the wait for a "start cycle" signal from the robot,
2. the machining of the part, and
3. the transmission of a "cycle finished" signal to the robot.

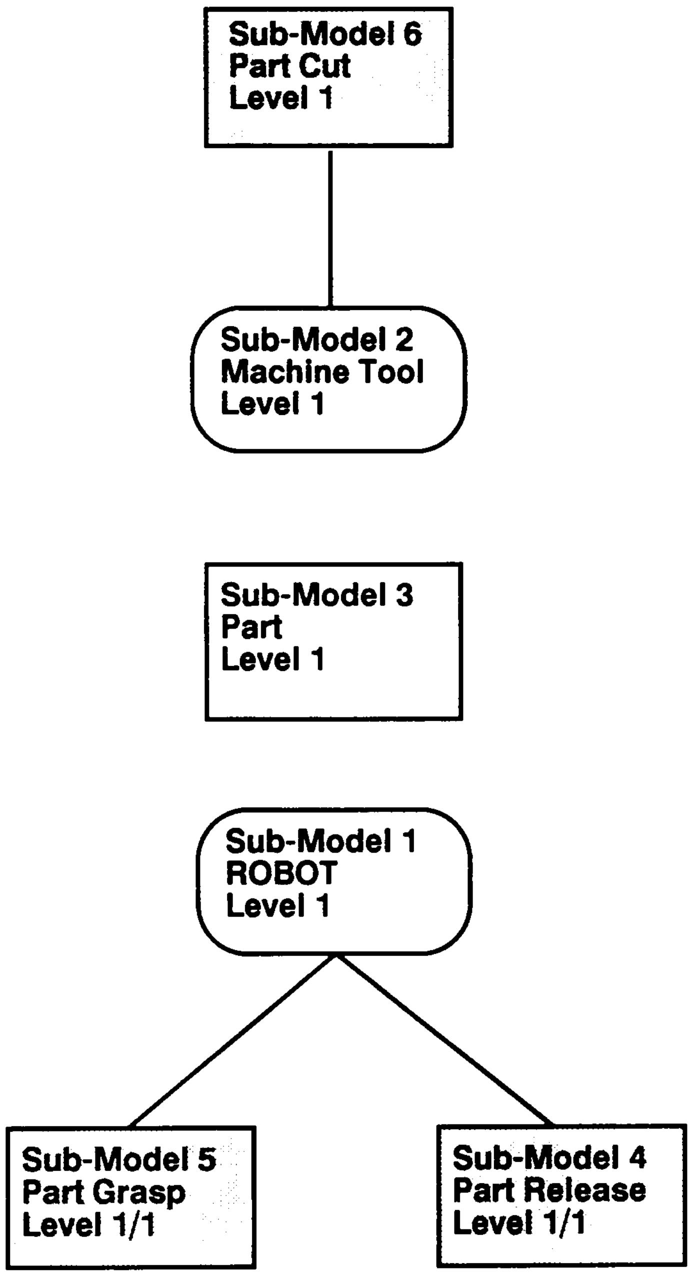


Figure 26 Level 1 Uni-Level Simulation Model

The part is represented with a passive sub-model. This sub-model has only one attribute, a part type identifier. Before processing, it has part type "A." After processing it has part type "B."

Associated with the robot, are two external interaction sub-models. Sub-model 4 monitors the beginning and completion of robot activity 1. When this activity ends, sub-model 4 transfers possession of the part sub-model from the robot sub-model to the machine tool sub-model. Likewise sub-model 5 is responsible for monitoring robot activity 4 and the transfer of possession of the part sub-model back to the robot sub-model.

Sub-model 6 is associated with the machine tool and monitors machine tool activity 2. When this activity ends, it changes the type identifier of the part from "A" to "B."

The lines drawn from the internal interaction sub-models to the external interaction sub-models indicate that the latter are dependent upon the initial execution of the former. To assemble the uni-level simulation sub-model from these sub-models, a modeler need only specify the internal interaction, external interaction and passive sub-models to be used and identify their dependencies.

Now lets consider another uni-level representation of the system, a kinematic/geometric model (see Figure 27). This model is considered a level 2 representation.

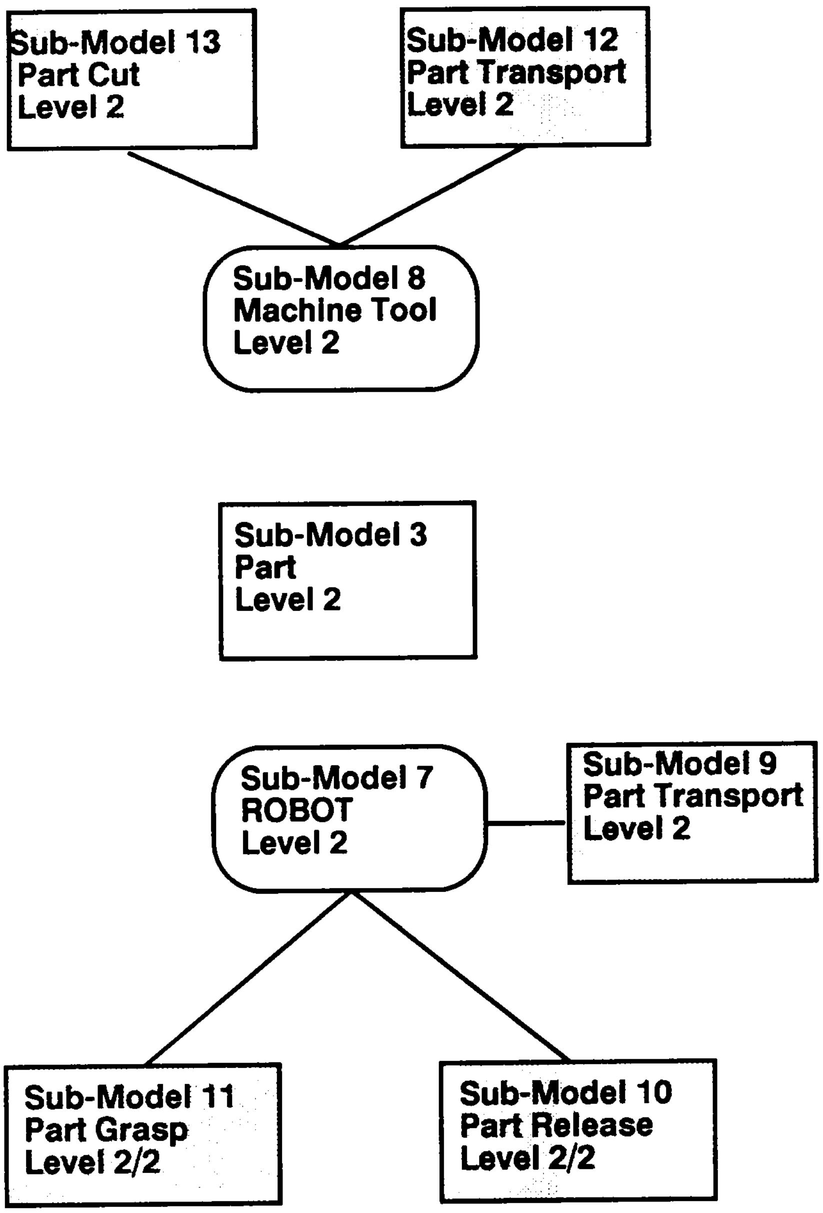


Figure 27 Level 2 Uni-Level Simulation Model

The robot and machine tool are again represented with internal interaction sub-models. The robot sub-model simulates the execution of a robot program and the subsequent actions of the robot manipulator. Kinematic attributes of the robot include the positions and velocities of the robot joints and links. Geometric attributes include the solids models of the links. Logical attributes include the robot program and its instruction pointer.

The simulated execution of the program involves the scanning of the following activities:

1. the transport of the part to the machine tool,
2. the release of the part from the gripper,
3. the movement of the manipulator away from the machine tool,
4. the transmission of a "start cycle" signal to the machine tool,
5. the wait for a "cycle finished" signal from the machine tool,
6. the movement of the manipulator back to the machine tool,
7. the grasping of the part by the gripper, and
8. the transport of the part away from the machine tool.

Movement of the robot manipulator is simulated through the firing of joint and link kinematic description functions over the passage of simulated time in conjunction with the scanning of the activities listed above.

The machine tool sub-model performs in a similar fashion. Machine tool program execution is simulated through the scanning of the following activities:

1. the wait for a "start cycle" signal from the robot.
2. the transport of the part away from the home position,
3. the transport of the part back to the home position, and
4. the transmission of a "cycle finished" signal to the robot.

The part is again represented with a passive sub-model. The kinematic attributes of this model are its position and velocity. Its geometric attributes are represented by a solids model.

There are three external interaction sub-models associated with the robot. Sub-model 9 monitors robot activities 1 and 8. During the execution of these activities, the sub-model simulates the transport of the part (possessed by the robot) by reading the kinematic attributes of the robot gripper and computing the kinematic attributes of the part. Sub-models 10 and 11 act in the same fashion as sub-models 4 and 5 in the previous example.

Sub-models 12 and 13 are associated with the machine tool. Sub-model 12 works in the same fashion as sub-model 9. Sub-model 13 monitors machine tool activity 2. During the simulation of this activity, the sub-model simulates the cutting of the part by subtracting the solids model associated with the machine tool cutter from the solids model associated with the part. Note again that lines drawn between sub-models indicate dependencies.

As a final example, lets consider a multi-level simulation of the system. This is illustrated in Figure 28. Here the discrete event representation of the robot is merged with the kinematic/geometric representation of the machine tool. Note

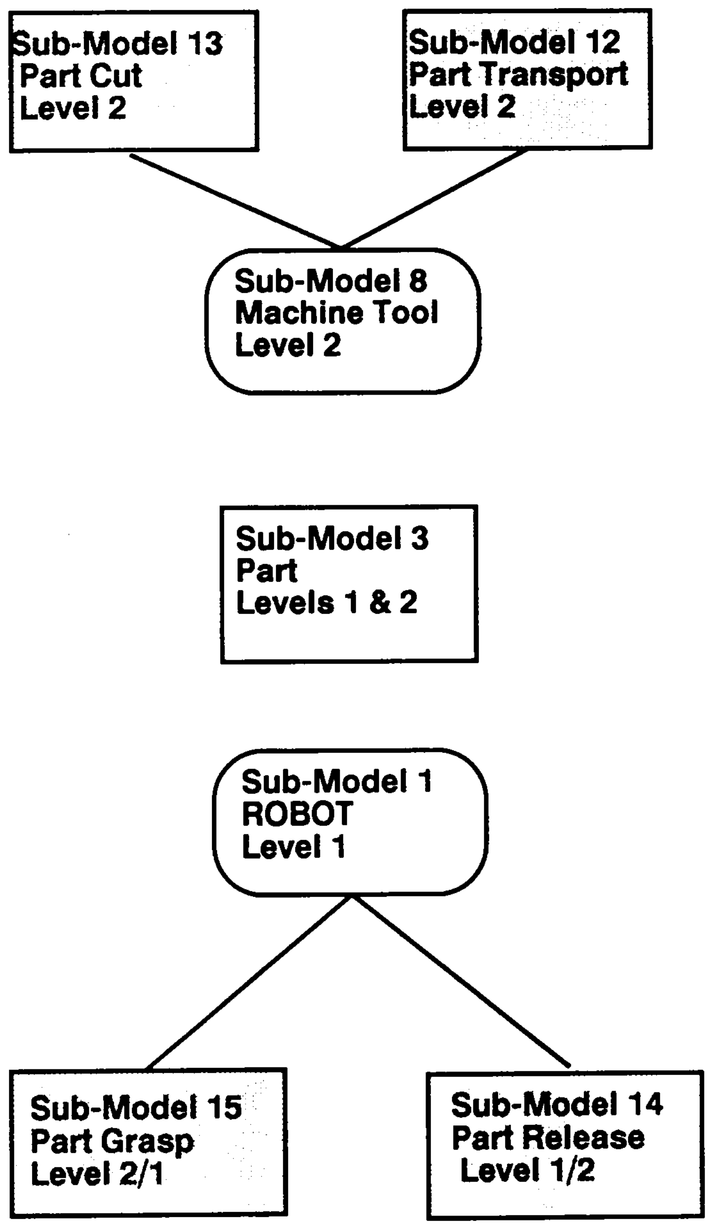


Figure 28 Multi-Level Simulation Model

that all but two of the internal and external interaction sub-models are the same as described in the previous examples. Only two new external interaction sub-models are needed to create this multi-simulation model. Before these new sub-models are discussed, the dynamic attribute structure of passive sub-models is re-addressed.

A dynamic attribute structure permits a passive sub-model to have its attribute set expanded or contracted during simulation execution. Thus the passive sub-model representing the part in the previous examples could potentially have a part type identifier at one instant of simulated time and kinematic/geometric attributes at another. The level of modeled interaction between a passive sub-model and its environment dictates which attributes the sub-model has. The entity which actually expands or contracts the set of modeled attributes is the external interaction sub-model associated with the interaction.

Getting back to the example, sub-model 14 monitors robot activity 1. When the activity ends, the sub-model adds kinematic/geometric attributes to the part sub-model, initializes their values, and then transfers possession of the part sub-model to the machine tool sub-model. Sub-model 15 monitors robot activity 4. When the activity ends, the sub-model reads the kinematic/geometric attributes of the part sub-model, determines which part type they correspond to, assigns a value to the part type identifier, discards the kinematic/geometric attributes, and transfers possession of the part sub-model back to the robot sub-model.

While not discussed in the previous examples, the importance of passive sub-model and internal interaction sub-model creation and destruction needs to be addressed. Passive sub-model creation is a common operation in most simulations utilizing GIBSS. A good example is the simulated arrival of parts into the system. Before they arrive, they are of no interest to the modeler, and thus are not allocated memory for attribute storage. Upon arrival into the system, the sub-models need to be created by the external interaction model simulating this process. Likewise, sub-models that leave the system are no longer needed and thus need to be destroyed.

The dynamic creation and destruction of internal interaction sub-models is a much rarer event. A modeler utilizes internal interaction sub-model creation when he desires to have a highly detailed sub-model replace a less detailed sub-model for some portion of a simulation. This replacement may ultimately call for the destruction of the old internal interaction sub-model or the new internal interaction sub-model after its use.

As an example of this process, consider the hypothetical dynamic/geometric simulation of the links and gripper of a robot within a work cell. For the majority of the simulation, the modeler may choose to maintain a dynamic/geometric sub-model of the gripper. However, if during the simulation it is anticipated that the gripper is to collide with another object, a more sophisticated model like a finite element/continuum mechanical representation may be created to replace the old gripper sub-model. This would permit the simulation of gripper deformation.

Note that the more sophisticated gripper sub-model could have been executed all along. But this would have increased the computation time of the general simulation, where as the detailed information that is provides is only required for a small simulated time segment.

Note that GIBSS does not alleviate the task of coding algorithms for internal interaction and external interaction sub-models. In fact, for any given simulation, a model created with GIBSS will implement the same types of algorithms and contain the same types of data as a model created with another frame work (assuming that it can be done). Alleviation of low level model specification and code creation is accomplished only through automatic model generation.

What GIBSS provides is a generalized frame work which enhances the code reusability of simulation models. A modeler, who structures a manufacturing system simulation model into passive, internal interaction, and external interaction sub-models and who uses the mechanical frame work discussed in the next section to create them, will have a set of sub-models which can be executed independently or in collective groups, validated independently or in collective groups, and linked together to form a comprehensive set of uni-level or multi-level simulation models. The next section discusses the simulation mechanics which permit this.

4.2 Generalized Mechanics of GIBSS

Under GIBSS, sub-models execute in parallel fashion when using a multi-process configuration and in psuedo parallel fashion when using a single process configuration. This section provides a generalized description of the main elements within GIBSS, discusses the structure of methods and data used to represent these elements, and describes the logic flow for both single process and multi-process implementations.

4.2.0 Functional Description of GIBSS Elements

GIBSS dictates that a simulation model be composed of simulation support entities and sub-models of various classes, which execute in parallel fashion. This is illustrated symbolically in Figure 29. Each of these components is discussed next.

4.2.0.0 Simulation Support Entities

Simulation support entities are data structures which support the simulation of a manufacturing system by storing data used for sub-model execution, synchronization, and communication. These entities are established and maintained in an accessible data segment called **GLOBAL STORAGE** along

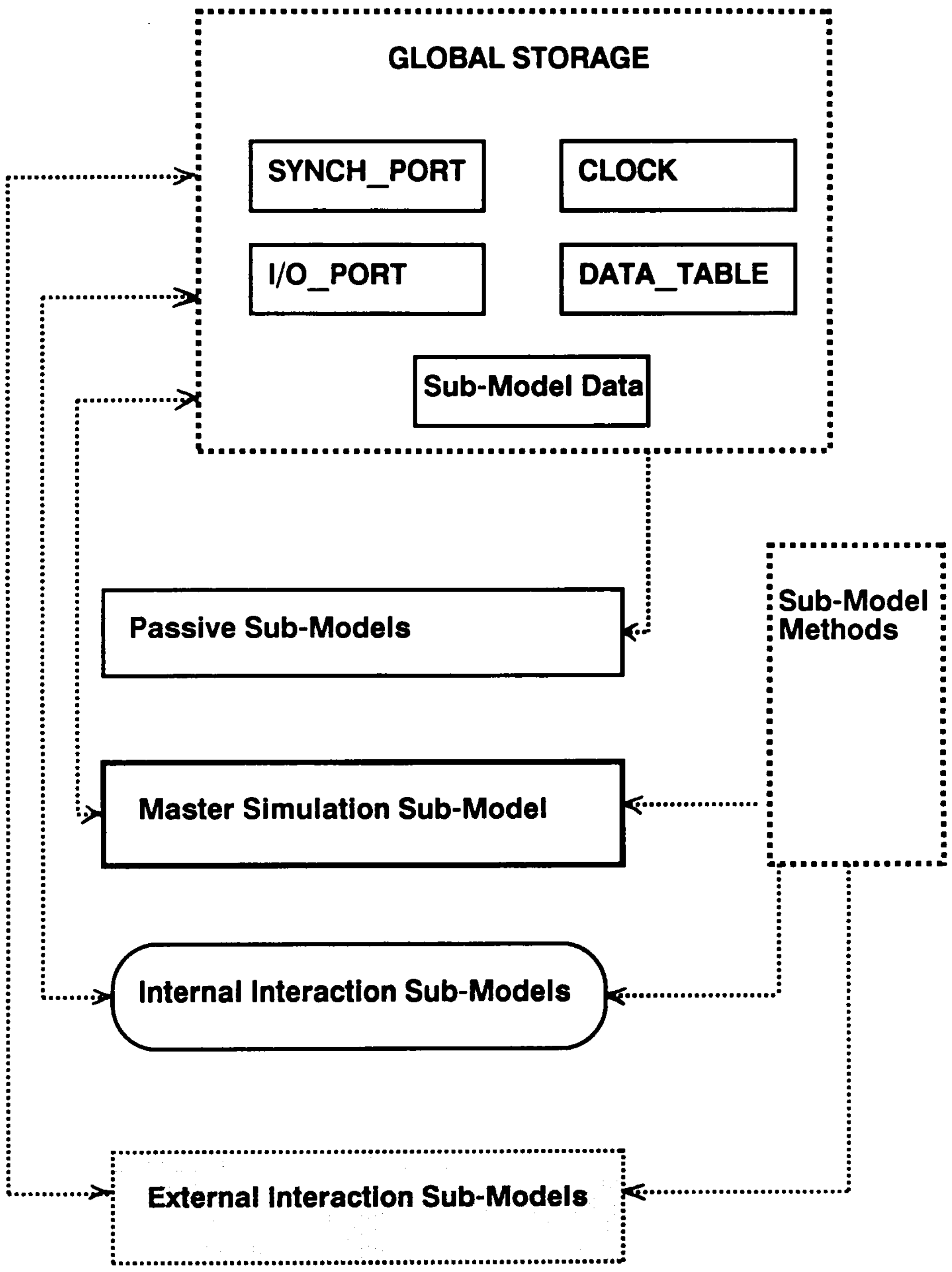


Figure 29 GIBSS Elements

with sub-model data. Accessible means that the all data within the segment are accessible to all methods used in the simulation model. The main support entities are **CLOCK, SYNCH_PORT, I/O_PORT,** and **DATA_TABLE.**

CLOCK is a simple data structure used to store the current time of the simulation.

SYNCH_PORT is a table which stores records used by sub-models to maintain synchronization. Each sub-model maintains a record in SYNCH_PORT. A record contains a minimum of two fields. One field stores a status flag, while the other stores a future occurrence time. The status flag indicates whether a sub-model is idle or currently simulating its sub-system. The future occurrence time indicates when a sub-model will begin an activity, end an activity, or fire an attribute description function.

I/O_PORT is a table which stores records used by internal interaction sub-models to simulate communication channels. Each record represents a port and contains a minimum of two fields. One field contains a transmitted message, while the other maintains the time at which the message was last updated. Each internal interaction sub-model may control multiple ports.

DATA_TABLE represents a set of tables used by sub-models to store pointers to sub-model prototypes, simulation initialization data, and a large assortment of simulation reference data. This representation scheme is often called a relational data base [31]. Sub-model prototypes are data which identify the data structures of various passive, internal interaction and external interaction sub-models. Simulation initialization data specify which sub-models are used in a simulation, the location of their prototypes, and the conditions for simulation termination. Simulation reference data provide cross references between attribute sets representing the same entities. For example, locations within a work cell could be represented by a table of symbolic identifiers and by a table of axial coordinate sets. To establish a simple means of cross reference, each location could utilize the same index for each table.

The manner in which these simulation support entities are manipulated during a simulation is described next.

4.2.0.1 Sub-Models

Sub-models are entities which utilize data and possibly methods to represent their respective sub-systems. There are four classes of sub-models:

1. passive sub-models,
2. master simulation sub-models,
3. internal interaction sub-models, and
4. external interaction sub-models.

These are discussed separately.

4.2.0.1.0 Passive Sub-Models

Passive sub-models are data structures used to represent functionally inert entities within the scope of a manufacturing system simulation. They utilize a dynamic set of attributes, which allow them to describe entities to various levels of detail through out a simulation. The structural properties which permit this set contraction or expansion are discussed in a subsequent section.

Passive sub-models do not have the ability to alter their own attributes and thus do not possess any simulation methods. Instead, their attributes are altered by external interaction sub-models. As a result, passive sub-model attributes are considered to be accessible to other sub-models.

Passive sub-models are considered either static or dynamic. Static sub-models represent entities which remain within in a manufacturing system during a time of study. They are created and initialized at the very beginning of a simulation, which means that initial attribute data must be created by a modeler and stored on a permanent medium. Once created, static sub-models are given a permanent location in GLOBAL STORAGE to store their data and may be altered by external interaction sub-models during simulation execution.

Along with attributes, a static sub-model maintains a buffer to store the locations of dynamic sub-models that it is currently interacting with. A static sub-model is said to **possess** a dynamic sub-model if the address of the dynamic sub-model is stored within its buffer. This is illustrated symbolically in Figure 30.

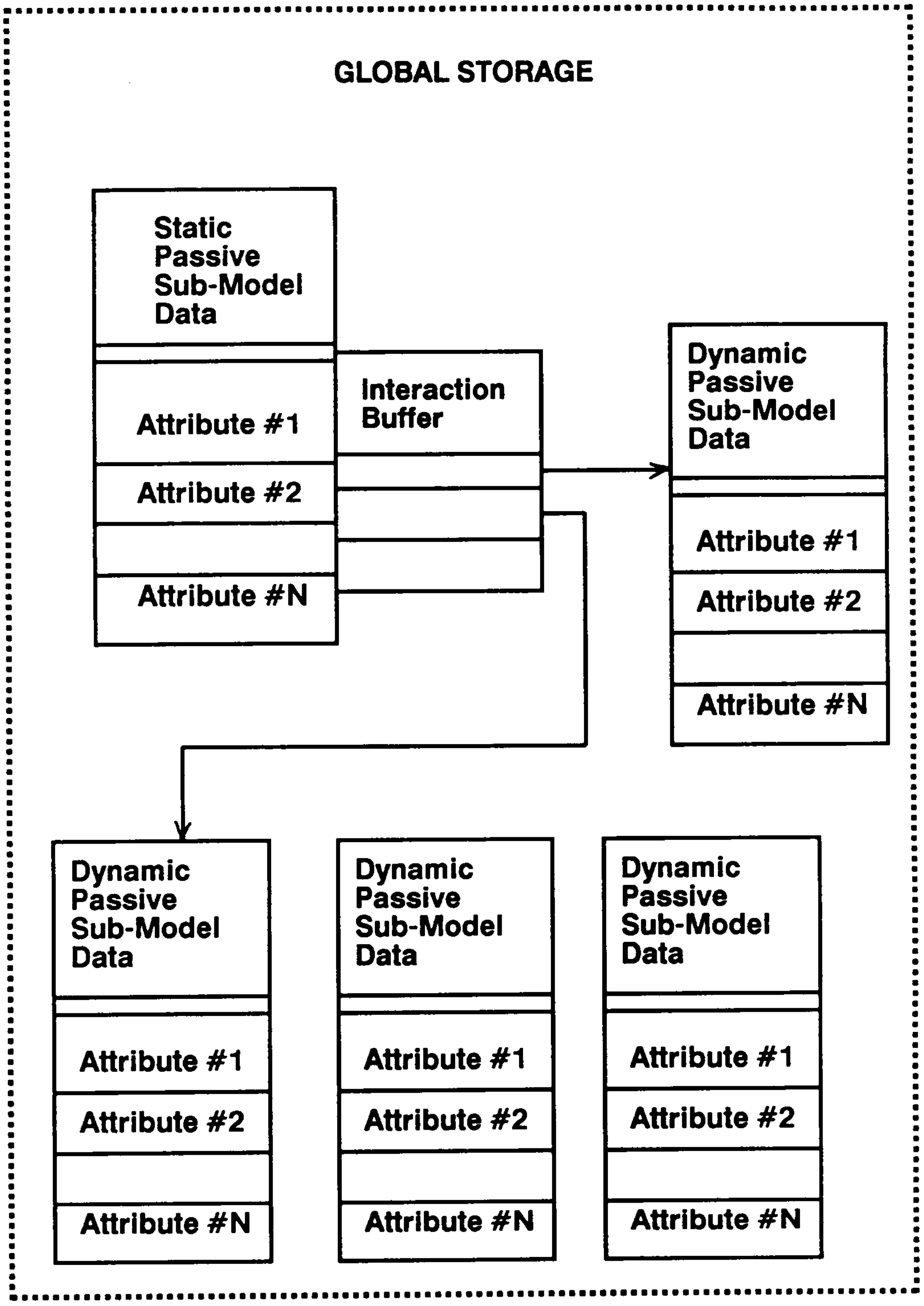


Figure 30 Interaction Buffers

Dynamic sub-models represent entities which exist temporarily within a manufacturing system during a time of study. The arrival of dynamic sub-models into a simulation is handled by external interaction sub-models. This arrival process involves finding a prototype of the dynamic sub-model in DATA_TABLE, allocating memory for its attributes in GLOBAL STORAGE, and storing its memory address in the buffer of a static passive sub-model or internal interaction sub-model.

Throughout their existence in a simulation, dynamic sub-models are possessed by the sub-models that they interact with or by sub-models that simulate their interactions with other sub-models. During their departure from a simulation, dynamic sub-models are destroyed by external interaction sub-models.

4.2.0.1.1 Master Simulation Sub-Models

The role of a master simulation sub-model is to initialize a simulation, synchronize the activities of internal interaction sub-models and external interaction sub-models, interact with a modeler if called to do so, and terminate a simulation when the proper conditions are met. Under GIBSS, a simulation model must utilize a master simulation sub-model, regardless of the number of constituent sub-models. In addition, a master simulation sub-model is the only sub-model which is directly invoked by a modeler during simulation execution.

The generalized logic flow for a master simulation sub-model is presented in Figure 31. During invocation, a master simulation sub-model enters the **simulation initialization** phase where it is passed the address of an initialization file. This file provides general initialization information as well the names of other element specific initialization files. The entire initialization file structure is arranged in a relational data base fashion to promote data modularity.

In general, the simulation initialization phase involves:

1. the reading of initialization data from an initialization file,
2. the creation of DATA_TABLE,
3. the spawning of internal and external interaction sub-models,
4. the initialization of internal and external interaction sub-model attributes,
5. the creation and initialization of static passive sub-models,
6. the storing of addresses of dynamic sub-model prototypes in DATA_TABLE,
7. the storing of general simulation reference data in DATA_TABLE,
8. the initialization of CLOCK,
9. the creation and initialization of I/O_ PORT, and
10. the creation and initialization of SYNCH_PORT.

The **clock update** phase involves setting CLOCK equal to the next future occurrence time and the setting of all status flags in SYNCH_PORT to busy. Upon CLOCK update, a master simulation sub-model enters the **sub-model scan** phase.

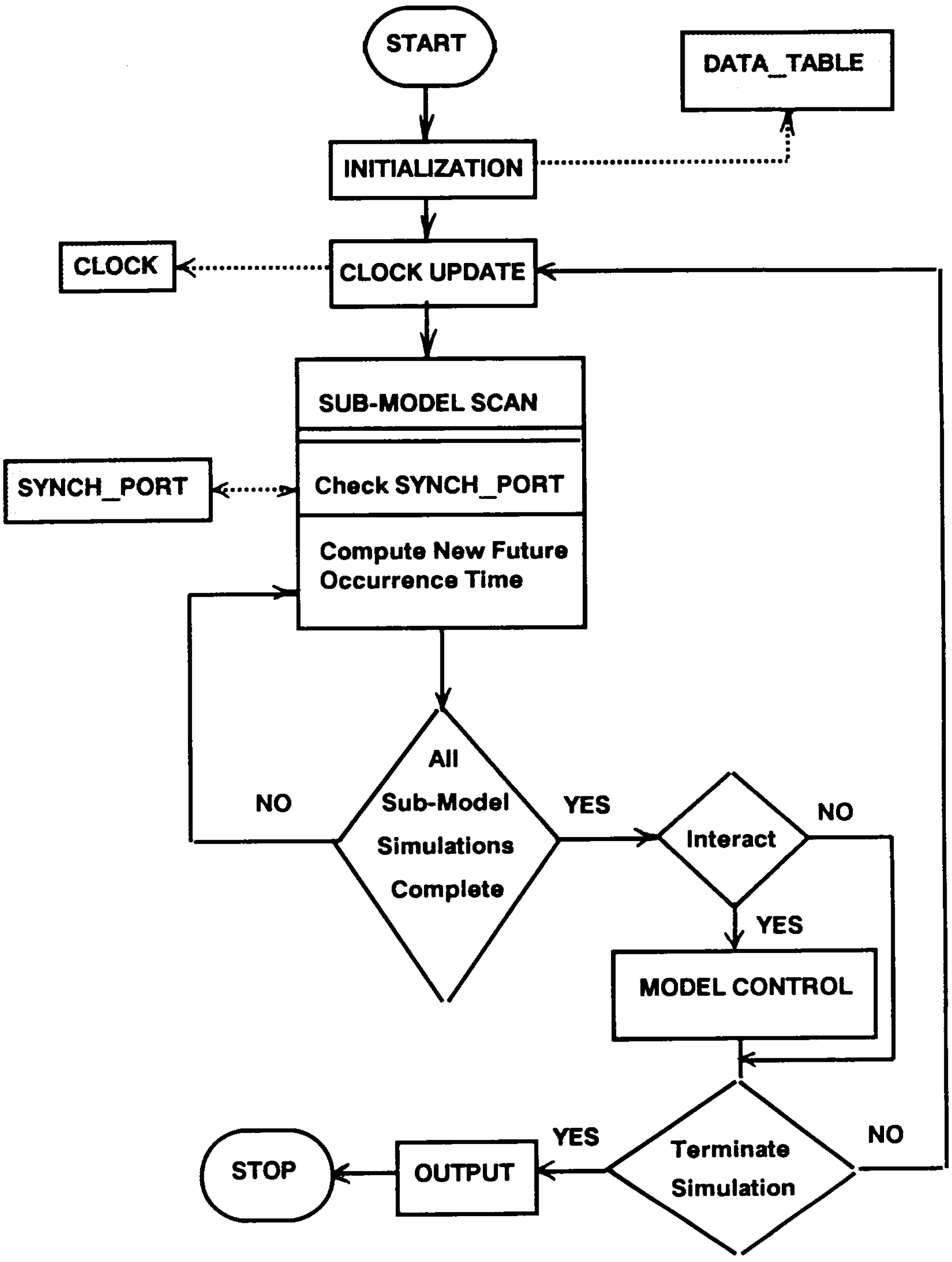


Figure 31 Master Simulation Sub-Model Logic Flow

This scan phase insures that all internal and external interaction sub-models complete simulation for the incremental change in CLOCK. In addition, the next future occurrence time for the simulation model is determined.

Monitoring the activities of independent sub-models is conducted through the repeated scanning of SYNCH_PORT. When sub-models have completed their individual simulations, they access their records in SYNCH_PORT, change their status flags from busy to idle, and record their future occurrence times.

When a master simulation sub-model detects that a sub-model has gone idle, it compares its future occurrence time to the smallest future occurrence time yet detected from any idle sub-model. If it is smaller, it becomes the value to which other future occurrence times are compared. Each sub-model utilizes its own time flow mechanism to track its own future occurrence time. In this fashion, sub-models are able to execute independently while the time flow mechanism of a simulation model remains independent of the time flow mechanisms of its constituents.

Once a master simulation sub-model is satisfied that all sub-models have become idle, it then determines whether a modeler requests interaction. If so, it enters the **model control** phase where it performs the actions requested by a modeler. Such actions may include the update of simulation parameters or the summarization of simulation data.

Upon completion of this phase, the sub-model checks for the conditions of simulation termination. If they are met, than the sub-model enters the **output** phase, where final simulation data is summarized and transferred to an output medium. If the conditions are not met, than the sub-model goes back to the clock update phase.

4.2.0.1.2 Internal Interaction Sub-Models

Internal interaction sub-models simulate the behavior of sub-systems by the scanning of activities (including ones of duration zero) and/or the execution of attribute description functions. With the exception of I/O_PORT communication, internal interaction sub-models monitor and change only their own attributes, and thus are independent of other internal and external interaction sub-models.

The generalized logic flow for internal interaction sub-models is presented in Figure 32. The **clock scan** phase involves monitoring CLOCK for a change in the current time of the simulation.

When a change is detected, the sub-model begins simulation by checking whether the new value of CLOCK is equal to its future occurrence time. If it is, the sub-model enters the **timed activity scan** phase, where it fires all of the time-based activity functions which correspond to the current value of CLOCK. If not, the sub-model skips directly to the **attribute function scan** phase, where it fires all of the appropriate attribute description functions for the incremental

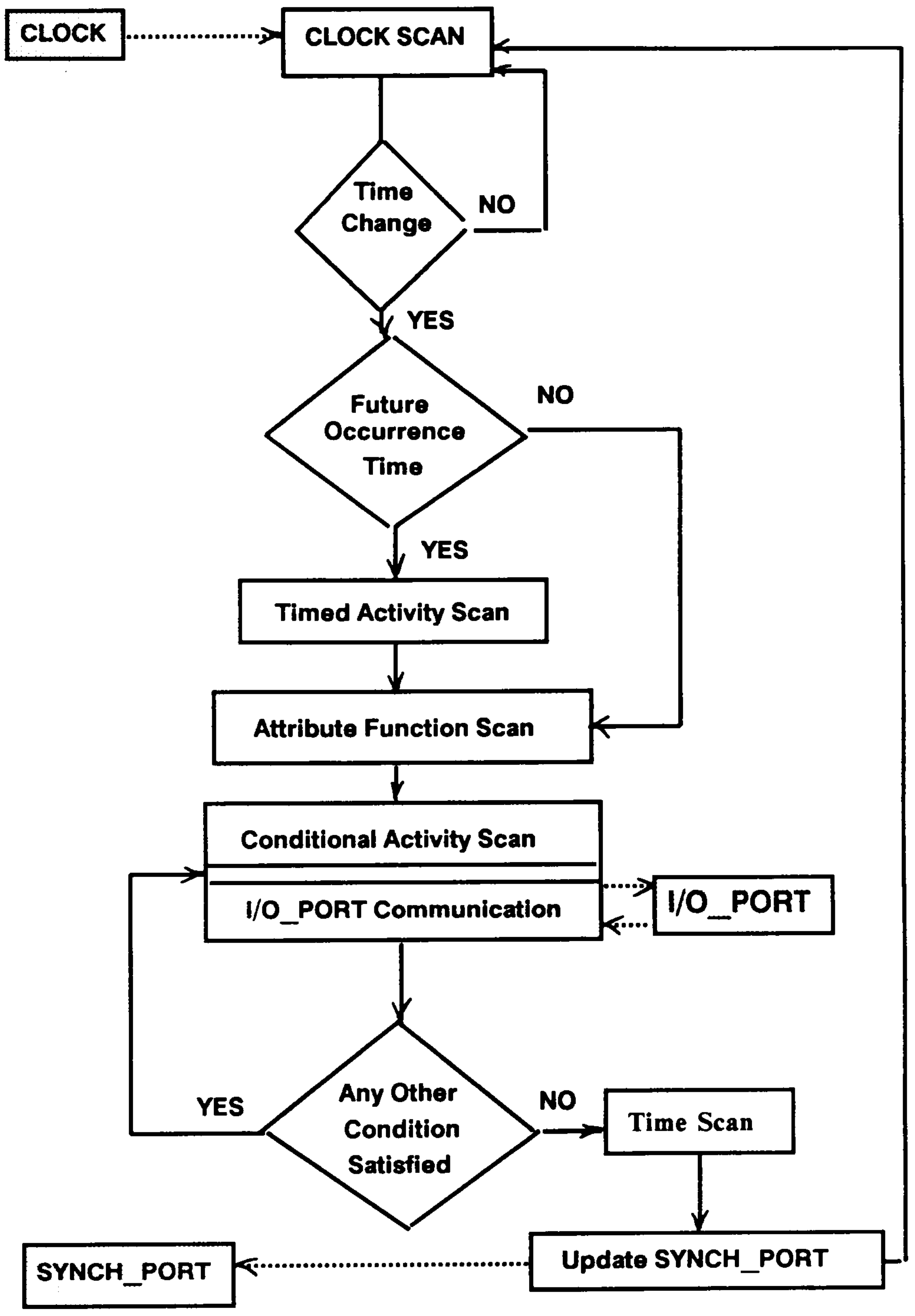


Figure 32 Internal Interaction Sub-Model Logic Flow

change in time. From here, the sub-model enters the **conditional activity scan** phase, where all conditional activities are repeatedly scanned until all fail to fire.

During the conditional activity scan phase, an internal interaction sub-model may simulate communication to other internal interaction sub-models by passing messages via I/O_PORT. In order for a sub-model to interpret a message from I/O_ PORT, the update time of the message must be equivalent to the current value of CLOCK. Otherwise the sub-model must wait until the I/O_PORT record is updated.

An internal interaction sub-model transmits a message by storing it in an appropriate record in I/O_PORT and updating the time of the port to the current value of CLOCK. Normally the transmission of a message occurs upon completion of the initial scan of the conditional activities. However during the occasion in which an internal interaction sub-model is waiting for the transmission of a message to an I/O_PORT record, it will prematurely transmit messages.

This phenomena commonly occurs when two internal interaction sub-models interactively communicate with one another. This is done in order prevent a condition known as **deadlock**. Deadlock occurs when the actions of at least two sub-models are conditionally dependent upon one another during an instant of simulated time, and results in the permanent stall of a simulation.

Note that an internal interaction sub-model is not required to utilize all of the three simulation phases just described. Instead, it can use any combination of the three as long as their order of execution is maintained.

Upon completion of its last simulation phase, an internal interaction sub-model determines its future occurrence time. How this is done is dependent upon which simulation phases the sub-model encompasses. If the sub-model executes only timed activities, then a variable increment time flow mechanism is used, and its future occurrence time is set equal to the completion time of its current activity or the beginning time of its next activity. If the sub-model only utilizes attribute description functions, then a fixed increment time flow mechanism is used, and its future occurrence time is set equal to the value of CLOCK plus its own fixed time increment. If the sub-model only executes conditional activities, then its future occurrence time is set equal to infinity (relative to the time of study). If the sub-model utilizes some combination of the three phases, then its future occurrence time is set equal to the smaller of the values just described.

Like dynamic passive sub-models, internal interaction sub-models utilize interaction buffers to store the addresses of dynamic sub-models that they interact with. The importance of this buffer is discussed in the next sub-section.

Once the methods and data of an internal interaction sub-model are created, it is linked to a simulation model by adding its methods to the general methods library of the simulation model and specifying in the simulation initialization file the location of its data in external storage, and the location of its corresponding records in SYNCH_PORT and I/O_PORT. It can then be executed in conjunction with the master simulation sub-model by itself or in parallel with other internal and external interaction sub-models. The selection of which sub-models execute is specified in the simulation initialization file.

4.2.0.1.3 External Interaction Sub-Models

External interaction sub-models simulate the interactions between sub-systems through the scanning of activities and/or the execution of attribute description functions. With the exception of the simulation of strictly time dependent behavior, external interaction sub-models are dependent upon either internal interaction sub-models or other external interaction sub-models.

The generalized logic flow for external interaction sub-models is presented in Figure 33. Note that it is very similar to the logic flow of internal interaction sub-models with some notable exceptions. After completing the clock scan phase, the sub-model must execute a **dependency scan** phase if it is dependent upon the initial execution of other sub-models.

During the dependency scan phase, an external interaction sub-model repeatedly scans SYNCH_PORT to check whether the sub-models that it is dependent upon have completed simulation. Upon exiting this phase, an external interaction sub-model then begins the simulation of interactions between other sub-systems. This simulation entails the execution of some

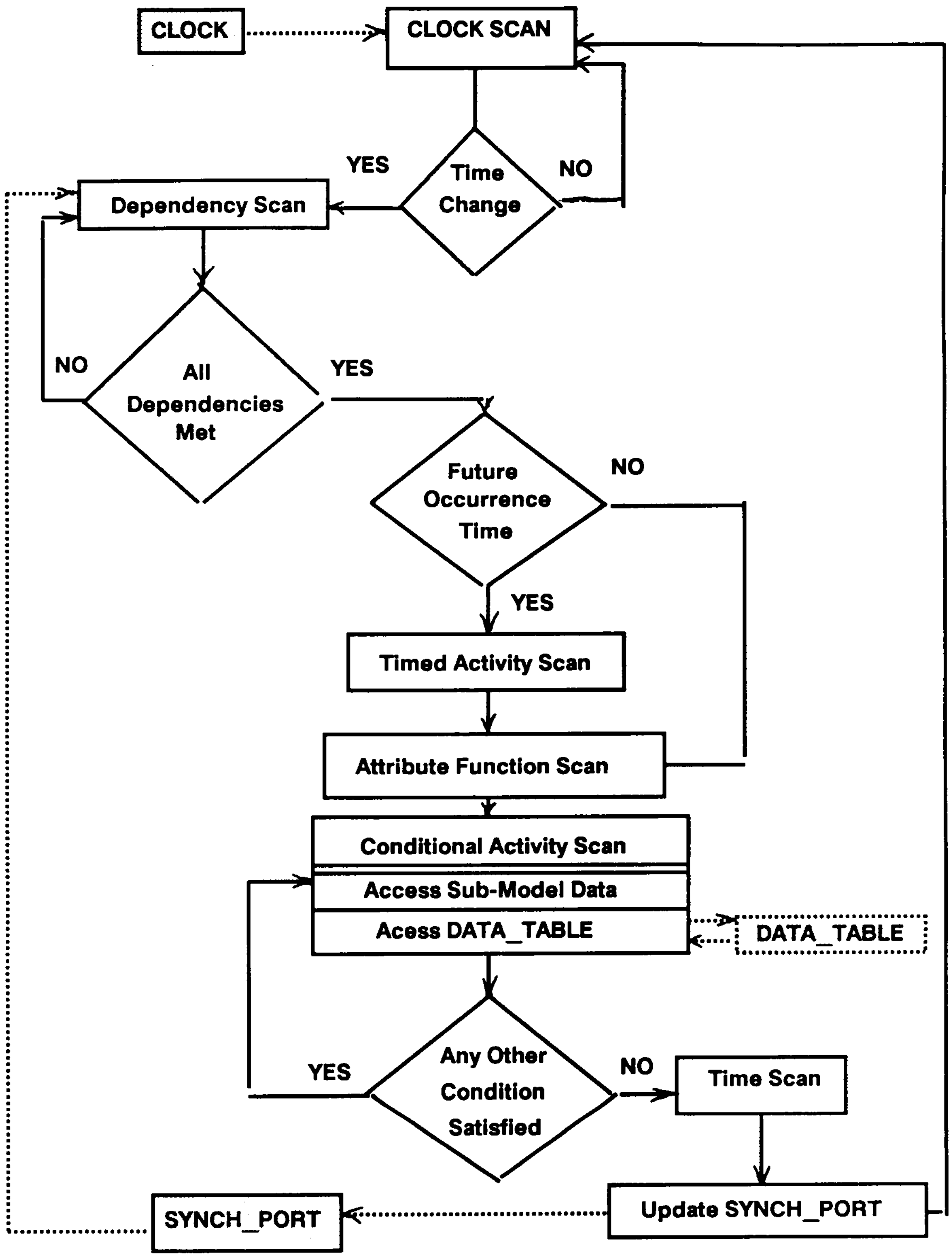


Figure 33 External Interaction Sub-Model Logic Flow

combination of the timed activity, attribute function, and conditional activity scan phases.

In order to simulate the interactions between sub-models, an external interaction sub-model must know where to find their attributes. Thus every external interaction sub-model maintains an interaction buffer(s). This buffer holds the addresses of sub-model data in GLOBAL STORAGE. These buffers are applicable to internal interaction sub-models, static passive sub-models, and possibly other external interaction sub-models. However dynamic passive sub-models are tracked in a different fashion.

As was discussed in sections 4.2.0.1.0 and 4.2.0.1.2, internal interaction sub-models and static sub-models typically possess interaction buffers, which are used to hold the addresses of the dynamic sub-models which they are currently interacting with. These buffers are utilized and maintained by external interaction sub-models responsible for simulating their interactions. Thus if an external interaction sub-model is responsible for simulating the kinematic interactions between a robot gripper and a part, it looks into its interaction buffer to find the address of the gripper sub-model, and looks into the interaction buffer of the gripper sub-model to find the address of the part sub-model.

In many cases, the responsibility of an external interaction sub-model is to transfer possession of a dynamic passive sub-model from one sub-model to another. This is accomplished by transferring the address of a dynamic sub-model from the interaction buffer of its current owner, to the interaction buffer of its next owner. Thus during a simulation, the addresses of dynamic sub-models are continually passed between various sub-models.

As was described in section 4.2.0.1.0, external interaction sub-models are responsible for the creation of dynamic passive sub-models and the expansion and contraction of their attribute sets. To create a dynamic passive sub-model, an external interaction sub-model locates the address of its prototype in DATA_TABLE along with its initialization data, allocates memory for the dynamic sub-model in GLOBAL STORAGE, and records the address of the dynamic sub-model into buffer of the internal interaction sub-model or static passive sub-model which it initially interacts with.

When expanding the attributes of a dynamic passive sub-model, an external interaction sub-model finds a prototype of the new class of attributes in DATA_TABLE, allocates memory for them in GLOBAL STORAGE, examines the current set of dynamic sub-model attributes, looks in DATA_TABLE for a cross reference between the current attributes and the new attributes, initializes the new attributes, and then assigns their address to the dynamic sub-model.

When contracting the attributes of a dynamic passive sub-model, an external interaction sub-model looks in DATA_TABLE for a cross reference between the current set of attributes and the future contracted set of attributes, initializes the contracted set, assigns their address to the dynamic sub-model, and then disallocates memory in GLOBAL STORAGE for the set of attributes no longer needed. GLOBAL STORAGE memory disallocation is also used to destroy passive sub-models. The same interactions with DATA_TABLE often take place for the creation, destruction, or modification of internal interaction sub-models.

Once the methods and data for an external interaction sub-model have been created, it is linked to an existing model by: storing its methods in the methods library of the model and by specifying in the initialization files where its data is to be found in external storage, the location of its corresponding record in SYNCH_PORT as well as those of the models upon which it is dependent, and the addresses of data that it needs to access in DATA_TABLE. It can then be executed in conjunction with the master simulation model by itself or in parallel with other internal and external interaction sub-models. The selection of which sub-models to execute is conducted through the simulation initialization file.

As a final point of interest, the framework for external interaction sub-models is also applicable to simulation data collection and summarization, since data collection is activity oriented. In the same manner that composite models are constructed, a modeler should be able to specify which types of simulation data are collected and summarized through the selection and addition of specialized data collection models to an existing composite model. This concept was not investigated in this dissertation, but will be investigated in future research.

4.2.1 Sub-Model Data and Methods Organization

GIBSS provides a generalized framework for the organization of a manufacturing system simulation model into various classes of sub-models. A major benefit of using GIBSS is increased model code reusability relative to the use of conventional simulation frame works. This benefit is further enhanced if a well organized approach is used for the implementation of sub-model data and methods. This research utilized concepts from the object oriented programming paradigm, **class data structures** and **methods libraries**, in the development of the test bed models.

Using this organizational scheme, GIBSS sub-models are created during simulation execution by the merging of sub-model data (organized under class data structures) with generic simulation methods stored in a methods library. This section discusses these concepts and how they are applied to the test bed models. Note that the development of organizational concepts for GIBSS sub-model data and methods was not in the scope of this research. However, the use of class data structures and methods libraries provided excellent results and is discussed for the sake of completeness.

4.2.1.0 Sub-Model Data Organization

During the development of the test bed models, large quantities of sub-model data were created, stored, and utilized. To organize this data, class data structures and data reference tables were utilized.

A **data structure** is a set of data that is organized under some framework. A data structure **type** dictates the organization of data for all data structures which are of its type.

A **class** data structure type has the characteristic that it not only dictates the organization of a set of data, but it also can point (contain the address of) to another class data structure type. Thus any class type data structure may be pointed to by a set of **super-classes** and may point to a set of **sub-classes**.

A class data structure can represent data not only associated with its own class, but with its sub-classes as well. Because of their extensive use of pointers, class data structures are capable of dynamically changing the amount of data that they represent. This is accomplished by the creation of class data storage through dynamic memory allocation and the destruction of class data storage through dynamic memory disallocation.

For the test bed model objects, the major class type is "**sub-model,**" followed by the sub-class types: "**master simulation sub-model**", "**internal interaction sub-model,**" and "**external interaction sub-model.**" These classes in turn have pointers to other sub-model class types.

Class data structures provide the characteristics which permit attribute set contraction and expansion. Passive sub-models are implemented using chains of class data structures. During attribute set expansion, memory is allocated for various class types in the chain, and their addresses are assigned to the pointers contained by their super-classes.

In general the positive attributes of class data structures are:

1. they provide well organized storage for data,
2. they allow easy access to large quantities of data through the simple passing of pointers,
3. they allow for the dynamic storage of data, and
4. they are easily linked together.

Data reference tables are generic arrays of data structures. These data structures may be complicated class data structures or simple integers or floats. The main characteristic of reference tables is their fast and easy access. A data structure within a table can be accessed by simply referencing its index value.

With respect to the test bed models, essentially all data contained in GLOBAL STORAGE during a simulation are organized into data reference tables. This includes all sub-model data, which are further organized into class data structures. As a result, tables are created to hold the data of such entities as static passive sub-models, dynamic passive sub-models, internal interaction sub-models, external interaction sub-models, and attribute classes, as well as pointers to files containing dynamic sub-model prototypes and initialization data.

4.2.1.1 Sub-model Methods Organization

To increase the resusability of simulation sub-model code, methods libraries are constructed. Methods libraries are subroutine libraries which are organized around the use class data structures. (the concept of subroutines is discussed in the next section). Every method within a library is created to accept the address of a class data structure and to operate on its data. All methods are generic in the sense that they are capable of operating on any data structure of a particular class type.

Methods within a library are organized into a hierarchical structure. At the bottom of the structure, are simple methods which operate only on simple class data structures. At the next level are more complex methods which operate on more complex data structures. These complex methods work by sending the addresses of sub-class data structures to lower level methods so that they may operate on them. This phenomena of increasingly complex methods which accept increasingly complex data structures repeats itself all the way up to the top of the library.

The positive attributes of methods libraries are:

1. they provide extensive code resusability due to their demand that all methods be generic with respect to class data structures,
2. they have easy access to large segments of data due their ability to accept the addresses of class data structures, and
3. their ease of expansion for reasons 1 and 2.

With respect to the test bed models, methods libraries are organized so that the methods at the top of the hierarchy handle the complete simulation of sub-models. Methods toward the middle portions of the libraries handle the simulations of sub-systems which are common to all sub-models, while methods at the bottom of the libraries handle basic computations.

Through out the development of the test bed models, the addition of new sub-model classes involved the creation of new class data structure types and methods. New class data structure types were created by linking existing class data structure types to the new class type, creating class specific data structure types, and linking the new class type to its super-class(s). New methods were developed by constructing a top level sub-model method and a few class specific methods, which relied primarily on the distribution of their tasks to methods already existing in the library.

Sub-models were created by specifying and storing sets of attribute data into files. Uni-level and multi-level simulation models were assembled by the creation of simulation initialization files, each of which specified the following:

1. the number of sub-models to be executed,
2. the location of sub-model data files and their type,
3. the allocation of records in I/O_PORT,
4. the allocation of records in SYNCH_PORT, and
5. the specification of DATA_TABLE locations for external interaction sub-models.

The relationship between class data structures, methods libraries, sub-model data, and composite model data is illustrated symbolically in Figure 34. More complete information on class data structures and methods libraries can be found in Martin [31] and Pascoe [39].

4.2.2 Single Process Implementation of GIBSS

In a single process configuration, all sub-model methods are implemented as subroutines, while GLOBAL STORAGE is implemented through the use of global variables.

Subroutines are executable code segments which are called from a main program or other subroutines. Execution of a subroutine by a CPU involves the

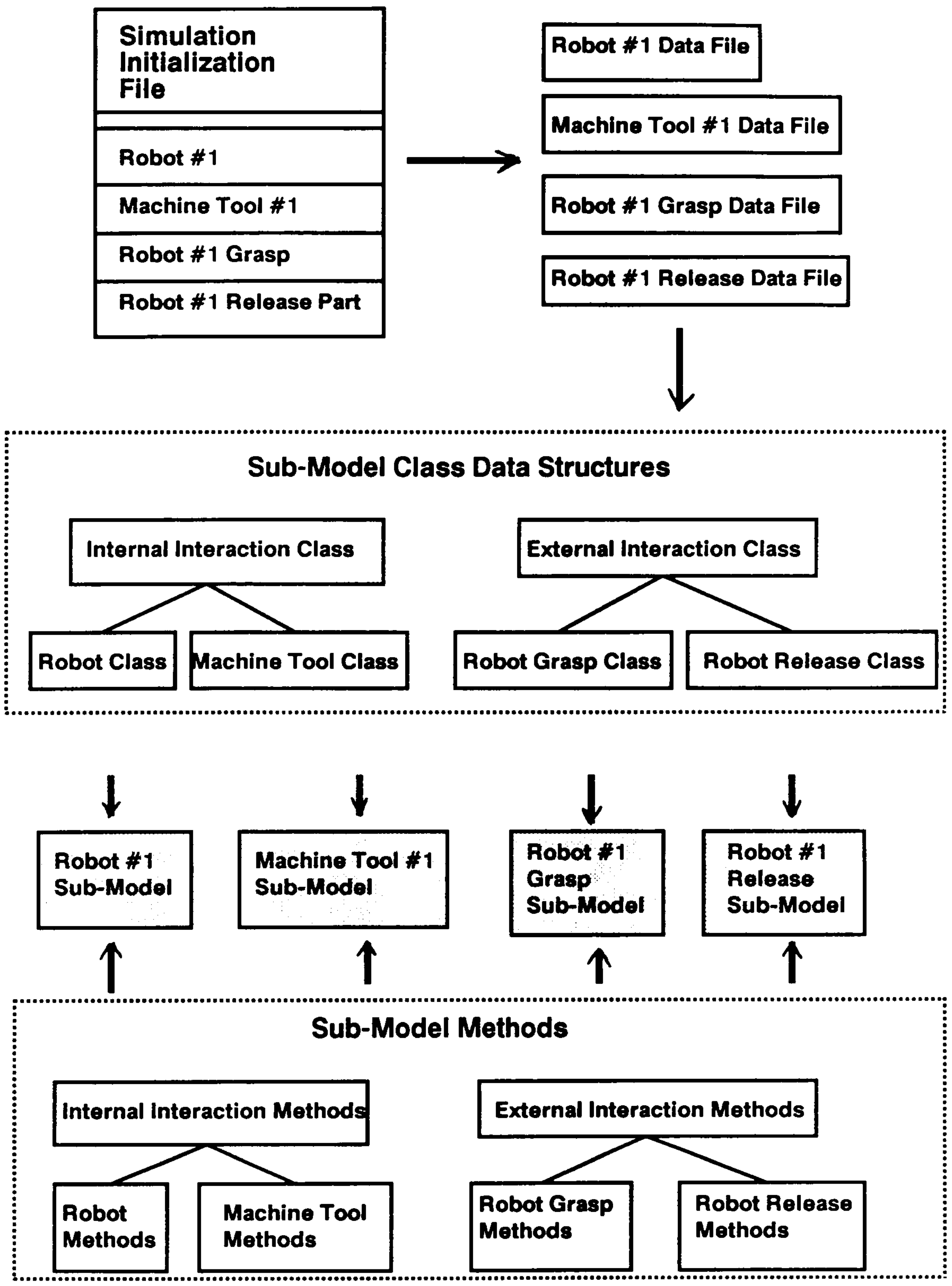


Figure 34 Sub-Model Data and Method Organization

passing of data (either by value or by address) from the calling routine to the subroutine, execution of the subroutine, and the return of CPU execution to the calling routine. This return process may also involve the passing of data from the subroutine to the calling routine.

A subroutine library is a segment of code containing multiple subroutines. Subroutine libraries are linked to an executable main program in order to provide it with access to the subroutines within the libraries.

Global variables are defined as those which are not **declared** relative to any one subroutine, but instead are accessible to all subroutines that form a single executable file. This is opposed to **local** variables which are only accessible by the subroutines under which they are defined.

The logic flow for a GIBSS simulation model using a single process configuration is shown in Figure 35. The main control of this simulation is provided by a master simulation sub-model, whose methods are implemented with either a main program or a high level subroutine. The simulation initialization phase is the same as described in section 4.2.0.1.1 with the exception that sub-models are not truly spawned. Instead their data are recorded into class data structures which are organized into a sub-model reference table.

The clock update phase involves a master simulation subroutine setting CLOCK to the next future occurrence time and pre-setting the status flags in

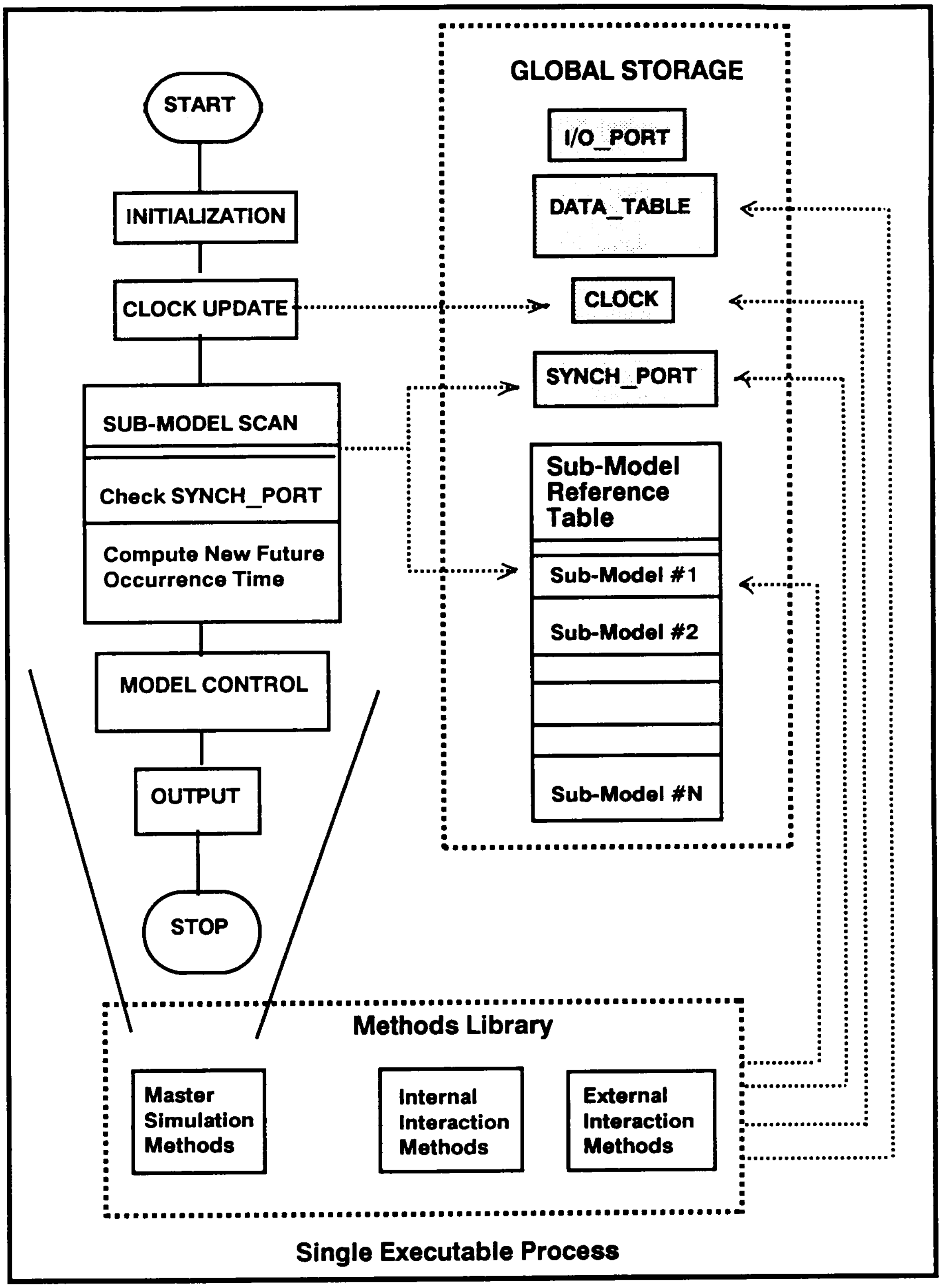


Figure 35 Single Process Logic Flow

SYNCH_PORT. The sub-model scan phase involves the execution of the individual sub-models. This is conducted by the master simulation subroutine sequentially referencing each sub-model data structure in the sub-model reference table and calling the simulation subroutine associated with its class. Sub-model data is provided to the subroutine by the passing of the reference table index.

After its initial scan through the sub-model reference table, a master simulation subroutine scans through SYNCH_PORT to determine if each sub-model has completed simulation. Upon finding a sub-model which has not, it then goes back to the sub-model reference table and calls the sub-model subroutine. This procedure is repeated until all sub-model simulations are complete.

The method of determining the next future occurrence time for the composite simulation as well as the model control and output phases are identical to those described for a master simulation sub-model in section 4.2.0.1.1.

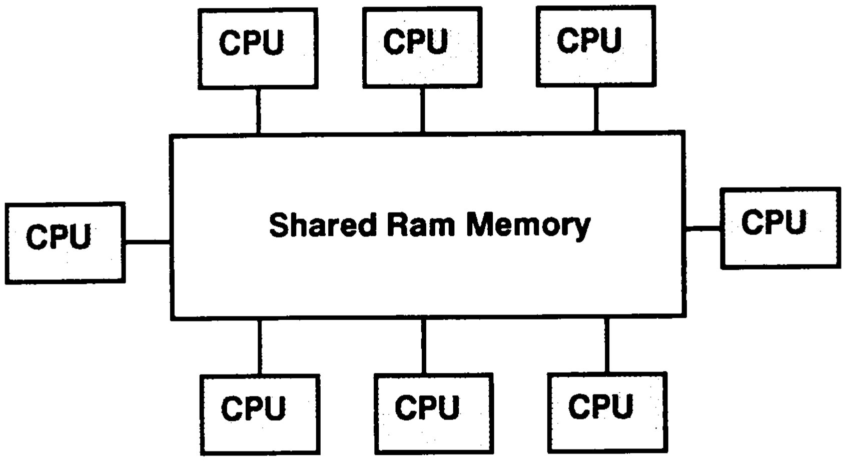
4.2.3 Multi-Process Implementation of GIBSS

Multi-process simulation relies on the cooperative execution of multiple processes to simulate system behavior. Processes were discussed in section 2.1.4.1. Executing a simulation in a multi-process configuration offers no advantages over a single process implementation unless it is conducted on a computer(s) utilizing multiple CPUs .

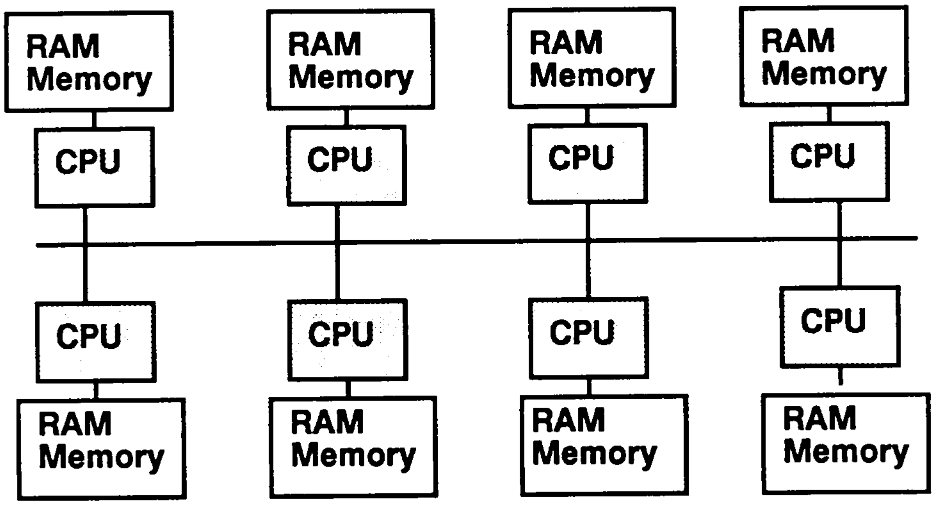
The manner in which processes work together on a computer(s) with multiple CPUs is highly dependent upon the architecture of the computer. The literature describes two common parallel architectures as the **shared memory** configuration and the **message passing** configuration [1]. In the shared memory configuration, CPUs are structured around a common segment of RAM memory. During program execution, processes executing on the individual CPUs communicate with one another by reading from and writing to variables stored in the shared memory segment. This is shown symbolically in Figure 36a.

In the message passing configuration, CPUs own their own segments of memory, which only they can access. The CPUs are connected by communication lines. These lines are either buses or serial communication lines. During program execution, processes executing on the CPUs operate on values in their own memory segments and communicate with one another by sending messages over their communication lines. This is shown symbolically in Figure 36b. This type of architecture was used by Concepcion and Schon [11] in the development of SAM.

Of the two architectures just described, the shared memory configuration is more suited to the implementation of GIBSS due to its heavy reliance on GLOBAL STORAGE. In a shared memory configuration, processes are used to implement the top level methods of sub-models, shared memory to contain GLOBAL STORAGE, and a **shared library** to house the composite methods library. This is shown symbolically in Figure 37 along with the logic flow of a



a. Shared Memory Configuration



b. Message Passing Configuration

Figure 36 Parallel Computer Architectures

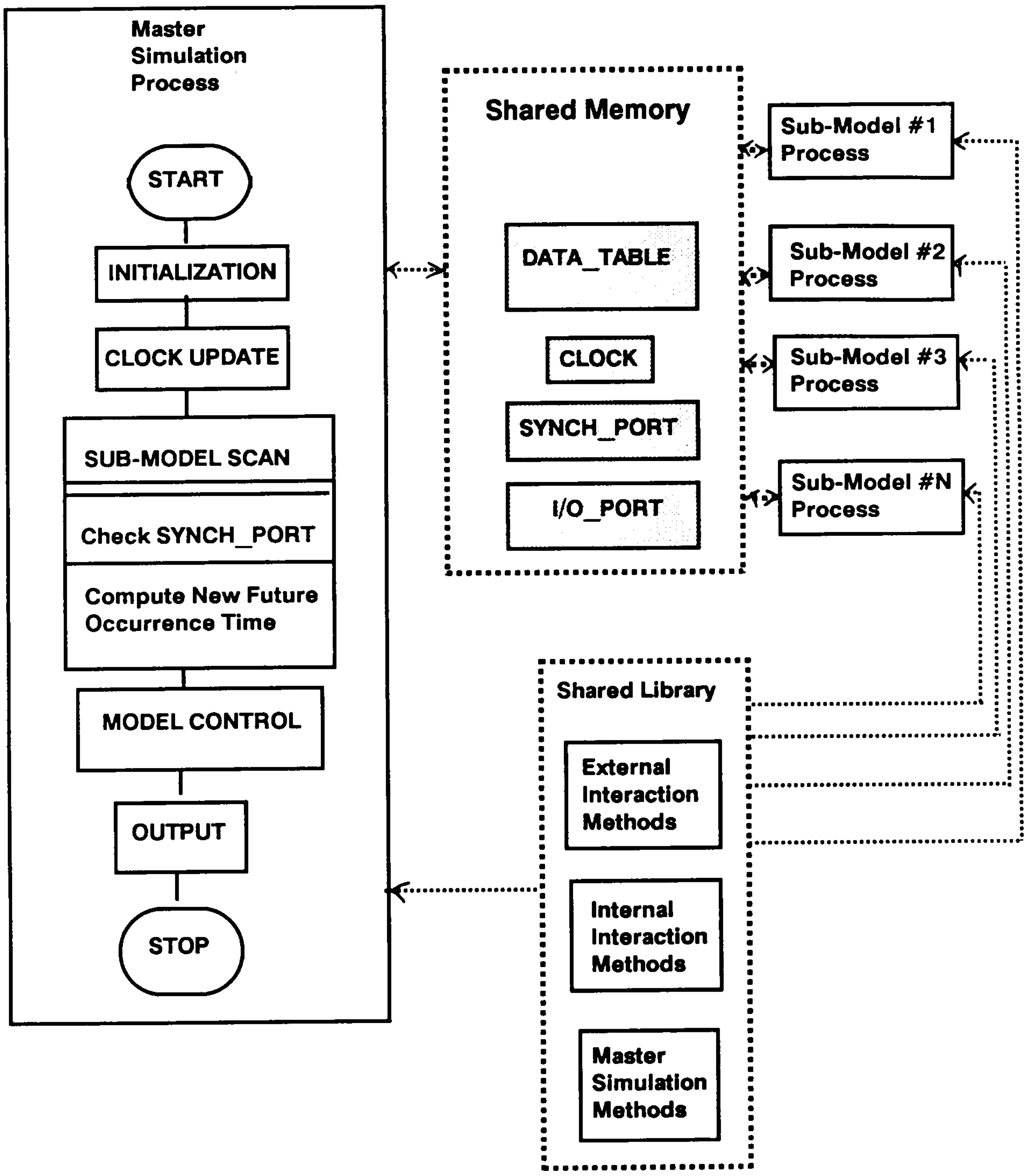


Figure 37 Multi-Process Logic Flow

multi-process implementation. Before this logic flow is discussed, a shared library is defined.

A shared library is a segment of executable code containing multiple subroutines, which is placed in RAM memory by an executing process, and which is accessible to all processes in RAM memory. If a GIBSS model were to be implemented with multiple processes without the use of a shared library, each sub-model process would have to own its own copy of the composite methods library. If the computer upon which the simulation is implemented has a large segment of memory, this poses no problem. However, if memory is limited and the number of sub-models great, then a shared library is needed.

The logic flow of GIBSS is divided among multiple processes. Main control of the simulation is provided by a master simulation process. The simulation initialization phase involves the spawning (invocation) of sub-model processes along with the other initialization operations described in section 4.2.0.1. The clock update phase is executed by the master simulation process and is identical to that of the single process implementation . During the sub-model scan phase, a master simulation process scans SYNCH_PORT and computes the next future occurrence time of the composite simulation, while the sub-model processes simultaneously simulate the manufacturing system. Each sub-model process follows the logic flow outlined for it in section 4.2.0. The model control and output phases are executed by the master simulation process and are identical to those in the single process implementation.

Implementation of GIBSS models using the shared memory configuration results in the same code modularity as the single process implementation. In addition, the parallel nature of GIBSS models is exploited, thus providing an avenue for reduced simulation execution time.

A multi-process implementation of the test bed models was originally pursued in this dissertation. While a UNIX PC does not have a parallel architecture, it does support the multi-tasking of processes as well as shared memory segments and shared libraries. This approach was abandoned for a single process implementation when it was learned that UNIX PCs lacked shared memory allocation routines. While these routines could have been created, it would have taken time from the overall research and not contributed to the generality of the framework. In the near future, these allocation routines will be created and GIBSS will be implemented in a multi-process configuration as extended research.

4.3 Summary and Conclusions

This chapter has described GIBSS, a generalized simulation framework which enhances the modular construction of uni-level and multi-level simulation models and which exploits parallelism within manufacturing system simulation models. GIBSS enhances model modularity by its requirement that the attributes and mechanisms of a simulation model be distributed among passive, internal interaction, and external interaction sub-models, and its specification of how these sub-models are linked to a master simulation sub-model to be executed independently or together in parallel fashion.

GIBSS is generalized with respect to manufacturing system simulation due to the logic flow of internal interaction and external interaction sub-models, which execute activity scans, event functions, and attribute description functions as well as their own time flow mechanisms. In addition, the logic flow of a master simulation sub-model insures that the time flow mechanism of a simulation model is always independent of the time flow mechanisms of its constituents.

GIBSS exploits parallelism within manufacturing system models by its specification of how a uni-level or multi-level simulation model is executed in a multi-process, shared memory configuration. In general, the execution of master simulation, internal interaction, and external interaction sub-models is distributed among multiple processes. GLOBAL STORAGE is implemented in shared memory, and the methods library is implemented in a shared library.

The contribution of this research to manufacturing system simulation has been the development of:

1. the concept of external interactions,
2. the concepts of master simulation sub-models, internal interaction sub-models, external interaction sub-models, and passive sub-models, and
3. the generalized mechanics for the sub-models just described.

To a much lesser extent, the application of class data structures and methods libraries in the test bed models has also provided a contribution since it has demonstrated an existing methodology which can further enhance the modularity of GIBSS models. The next chapter describes the model demonstration environment used to demonstrate GIBSS concepts.

5 FRAMEWORK DEMONSTRATION

5.0 Introduction

To demonstrate GIBSS, an interactive model demonstration environment has been created. This environment allows a user to select from twenty-five GIBSS models, view their sub-model structure, and witness their execution through the aid of graphical animation. This chapter discusses the nucleus of the environment as well as its operation.

5.1 Environment Nucleus

The model demonstration environment utilizes the uni-level and multi-level test bed models developed in this dissertation. These models represent the hypothetical manufacturing system described in section 3.3.0. In all, eighteen sub-model classes serve as the basis for the test bed models. The sub-model classes and their hierarchical structures are shown symbolically in Figure 38. A

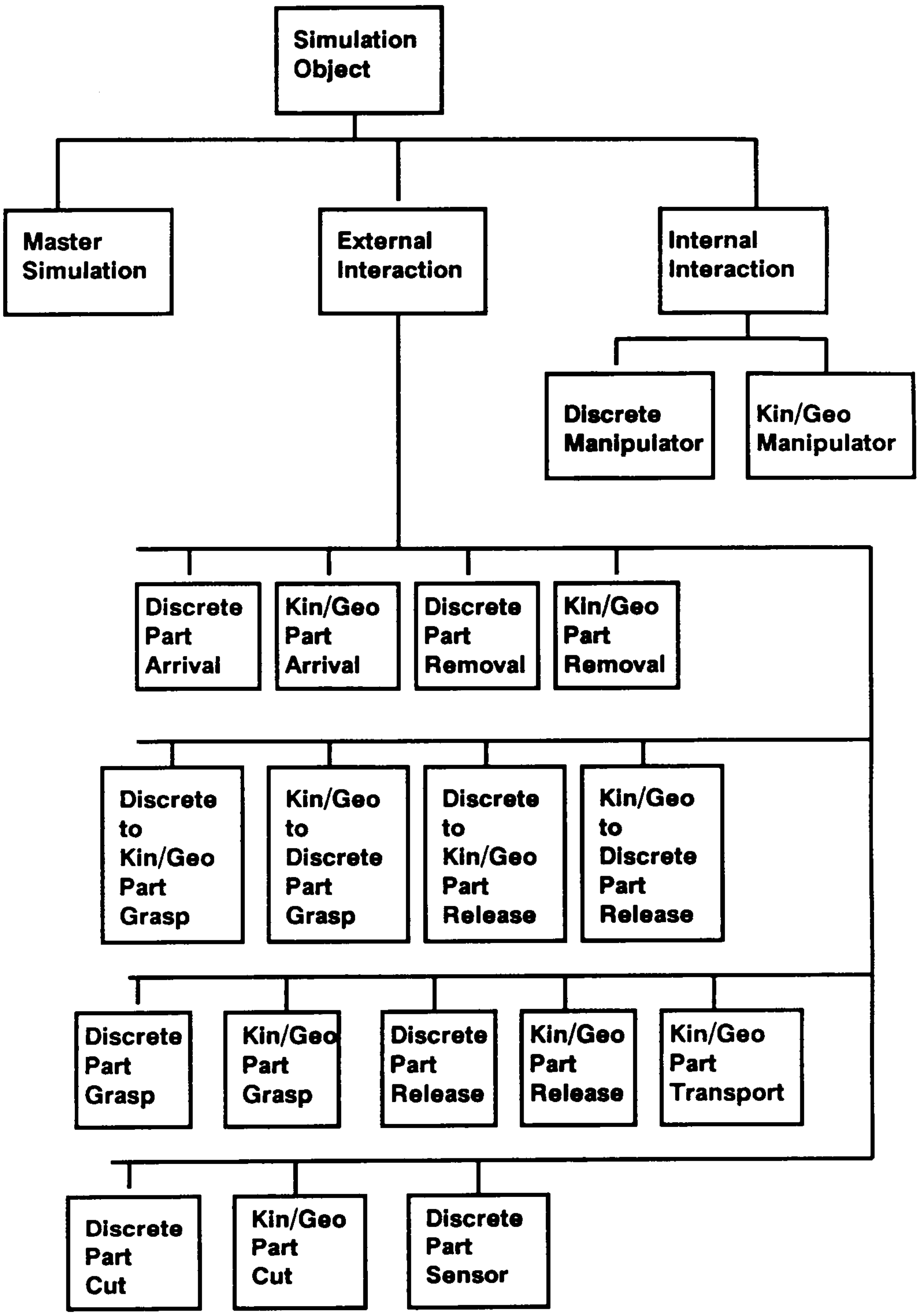


Figure 38 Sub-Model Class Structure

description of these sub-model classes is provided in Appendix B. Including the geometric modeling routines, the simulation methods entail roughly 15000 lines of C code.

Data was generated to create thirty-nine internal and external interaction sub-models, three static sub-models, and one dynamic sub-model prototype. The sub-models are used to create three uni-level simulation models and twenty-two multi-level simulation models. A description of the sub-models is provided in Appendix C. The structures of the twenty-five composite models are illustrated in Appendix D. The source code for both methods and class data structures is located in the Manufacturing Systems Software Library along with the sub-model data files.

5.2 Environment Operation

The logic flow for the model demonstration environment is provided in Figure 39. Upon invocation, the environment enters the **model selection** mode and displays twenty-five icons representing the twenty-five test bed models. The user is then prompted to select one of the twenty-five models. After the user types his selection, the environment symbolically displays the sub-model structure of the model in a fashion similar to that of Appendix D.

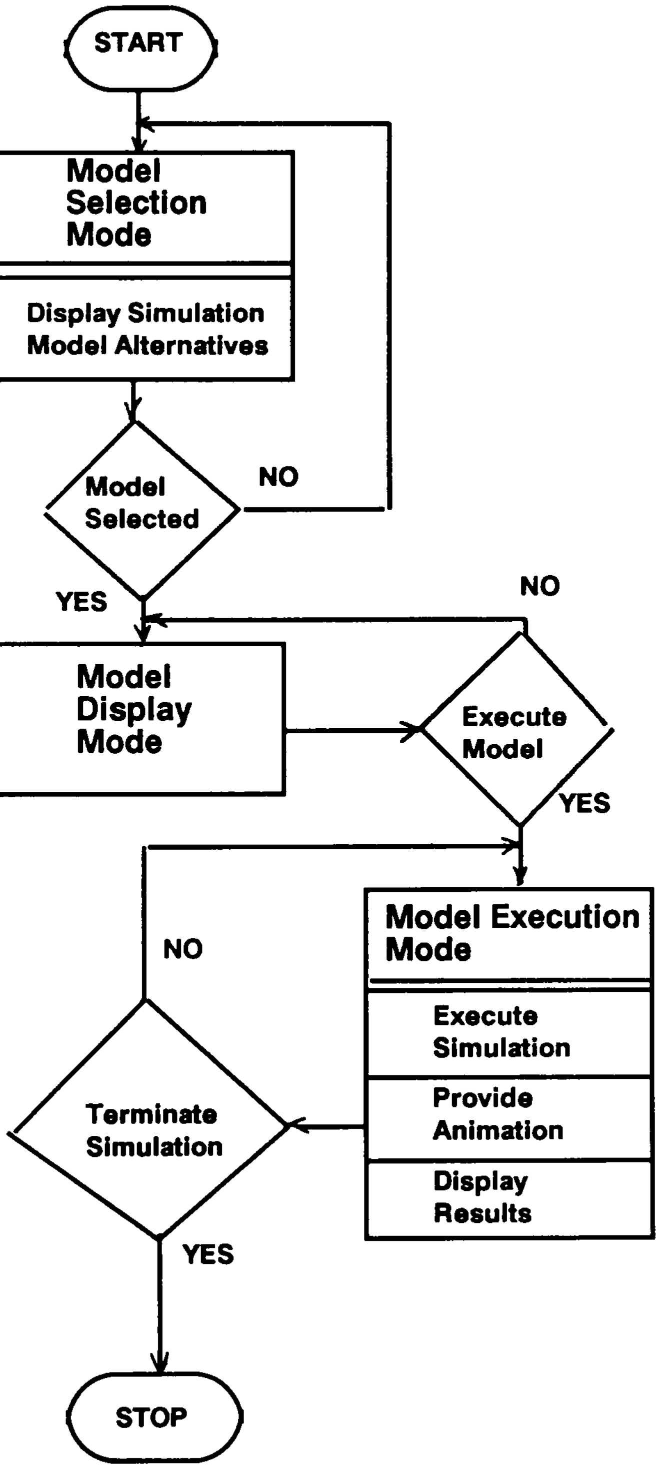


Figure 39 Demonstration Environment Logic Flow

Once the user has inspected the content of the model, he can then request the environment to enter the **model execution** mode. From here, the user can either begin model execution or re-enter the model selection mode. By repeating this procedure, the user can inspect the sub-model structures of many models and gain an appreciation for the reusability of the model code.

While in the model execution mode, the user controls the execution of a model by:

1. allowing the model to execute to a default simulation time limit,
2. stepping the advance of the simulation clock by each subsequent future occurrence time, or
3. setting a breakpoint in simulated time at which model execution suspends.

Model execution is accompanied by a graphical animation of the simulated manufacturing system. A snap shot of the animation is presented in Figure 40. The animation displays the value of the simulation clock along with a set of stationary rectangles which represent sub-models that simulate a combination of the following entities:

1. work cell 1,
2. work cell 2,
3. robot 1,
4. robot 2,
5. machine tool 1,

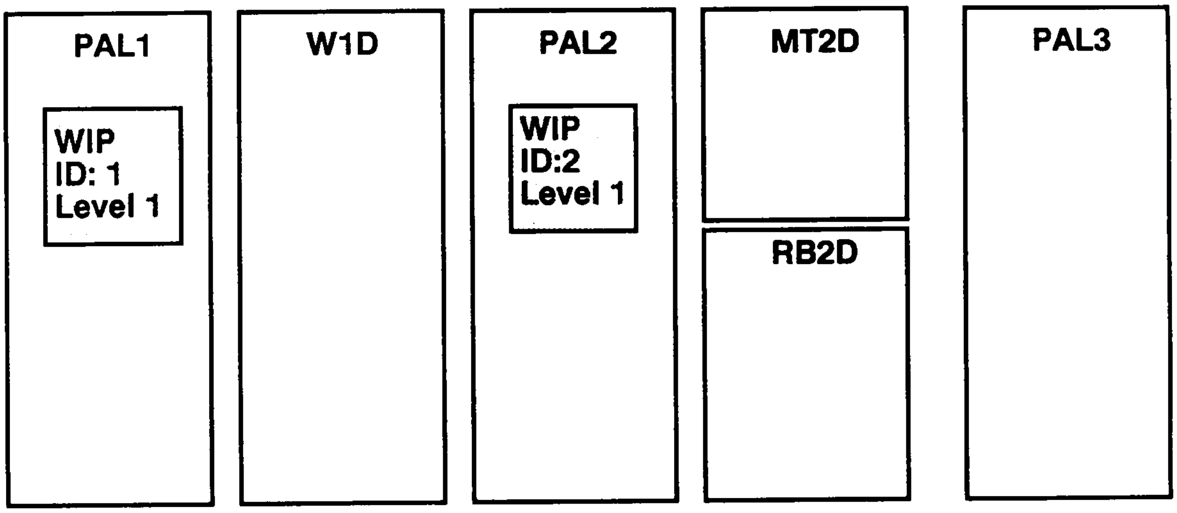


Figure 40 Snap Shot of Environment Animation

6. machine tool 2,
7. pallet 1,
8. pallet 2, and
9. pallet 3.

Each stationary rectangle has a label to indicate the entity that it represents, and a suffix to indicate the level of representation. A **D** suffix indicates a discrete event representation while a **K** suffix indicates a kinematic/geometric representation.

A dynamic set of rectangles, which represent part sub-models, appear on the monitor and move between the stationary rectangles. Each dynamic rectangle has a label for part identification and a label for attribute set identification. **Level 1** indicates that the attribute set includes the symbolic type and location of the part. **Level 2** indicates that the attribute set includes the position and velocity of the part as well as its solids model representation. When ever a dynamic rectangle appears within a stationary rectangle, it indicates that the sub-model currently possesses the part sub-model.

Of potential interest to the user is the animated display of a multi-level simulation. During model execution, the expansion and contraction of part sub-model attribute sets is illustrated as possession of the part sub-models is transferred between other sub-models. By suspending model execution, the user can inspect the values held by these sets along with the attributes of the other displayed sub-models. To do this, the user chooses the sub-model from a menu, and then chooses which attributes he wishes displayed.

In conclusion, the model demonstration environment is fairly simplistic. It does not utilize shaded graphics or sophisticated wire frame images. However it is sufficient to provide a user with a better understanding of GIBSS and the concepts of uni-level and multi-level simulation. The next chapter summarizes this dissertation, draws conclusions, and discusses future research.

6 SUMMARY AND CONCLUSIONS

6.0 Summary

The multi-level design of manufacturing systems was discussed. It was concluded that to facilitate the design process, models of sub-designs must be applicable to modular assembly, even if the sub-designs are heterogeneously specified.

Computer simulation was chosen as the technique for modularization due to its small mechanism set and its large problem domain. Two types of simulation models were identified: uni-level and multi-level. A model of a manufacturing system is considered uni-level if objects of equivalent type are modeled at the same level of detail. A model is considered multi-level if objects of equivalent type are not modeled at the same level of detail. It was concluded that current simulation frameworks do not integrate modular construction with the discrete event and continuous simulation techniques needed to support multi-level modeling.

The objective of this dissertation was to develop a simulation framework which could support the modular construction of uni-level and multi-level simulation models of manufacturing systems. It was recognized that to meet this objective, the simulation framework would have to exhibit a large degree of generality and modularity. Generality is defined as the ability of the framework to execute all of the known mechanisms used in the simulation of manufacturing systems. These mechanisms include the scanning of activities, the execution of event functions, and the execution of attribute description functions, as well as the utilization of both fixed and variable increment time flow mechanisms. Modularity is defined as the ability of the framework to allow a model to be divided into sub-models, each of which is created, executed, and validated independently, and then linked together to form an executable model.

In addition, it was recognized that the framework should exploit parallelism within manufacturing system simulation models in order to take advantage of parallel processing technology, and thus reduce the execution time of large scale simulations in the future.

This dissertation proceeded to describe GIBSS (Generalized Interaction Based Simulation Specification), a simulation framework designed to exhibit the characteristics just described. GIBSS requires that the mechanisms within a simulation model be distributed among a set of independent sub-models. These sub-models are classified as passive, internal interaction, or external interaction. In addition, GIBSS specifies the mechanics of each and describes how they work together in conjunction with a fourth class of sub-model, master simulation.

Through example and by a description of their mechanics, the generality of the sub-model classes was demonstrated along with their ability to support the modular construction of uni-level and multi-level simulation models. The ability of GIBSS to exploit parallelism was demonstrated through a description of how the sub-models work together in a multi-process, shared memory configuration. In addition, the merits of using class data structures and methods libraries in the implementation of GIBSS sub-models was also discussed.

Finally a simple model demonstration environment was described. This environment utilizes test bed models developed in this dissertation to illustrate the concepts of uni-level and multi-level simulation as well as the characteristics of GIBSS.

6.1 Conclusions

Five conclusions drawn from this dissertation are:

1. GIBSS promotes the modular construction of uni-level and multi-level simulation models,
2. GIBSS is generalized with respect manufacturing simulation,
3. GIBSS exploits parallelism within manufacturing system simulation models,
4. GIBSS eliminates a barrier to the rapid evaluation of manufacturing systems, and

5. GIBSS supports the multi-level design of manufacturing systems.

GIBSS promotes the modular construction of uni-level and multi-level simulation models through its requirement that the attributes and mechanisms of a simulation model be distributed among passive, internal interaction, and external interaction sub-models, and its specification of how these sub-models are linked to a master simulation sub-model to be executed independently or together in parallel fashion.

GIBSS is generalized with respect to manufacturing system simulation due to the logic flow of internal and external interaction sub-models, which execute activity scans, event functions, attribute description functions as well as their own time flow mechanisms. In addition, the logic flow of a master simulation sub-model insures that the time flow mechanism of a simulation model is always independent of the time flow mechanisms of its constituent sub-models.

GIBSS exploits parallelism by specifying how uni-level or multi-level simulation models are executed in a multi-process, shared memory configuration. In general, the execution of master simulation, internal interaction, and external interaction sub-models is distributed among multiple processes. GLOBAL STORAGE is implemented in shared memory, and a methods library is implemented through a shared library.

GIBSS eliminates a barrier to the rapid evaluation of manufacturing systems through its support of modular model construction, which promotes model reusability, and eliminates the need for model reconstruction or code duplication.

GIBSS supports the multi-level design of manufacturing systems due to conclusions 1 -4.

This dissertation has contributed to manufacturing simulation research by its development of:

1. the concept of external interactions,
2. the concepts of passive, master simulation, internal interaction, and external interaction sub-models,
3. the generalized mechanisms for the sub-models, and
4. the specification of how the sub-models are implemented in single process and multi-process configurations.

To a much lesser extent, this dissertation has made a contribution by its verification of the effectiveness of class data structures and methods libraries for enhancing the code reusability of GIBSS sub-models.

With respect to manufacturing system design, this research has made a contribution by its:

1. demonstration of how uni-level and multi-level simulation models are constructed and assembled during the multi-level design of a manufacturing system,

2. demonstration of how uni-level and multi-level simulation models are used to evaluate sub-designs, and uni-level and multi-level aggregate designs, and,
3. its elimination of a barrier to the rapid evaluation of manufacturing systems.

6.2 Future Research

GIBSS is the beginning of a long research effort directed toward the development of a new generation of computer-aided manufacturing system design environments. It is anticipated that these environments will support the multi-level design process by aiding designers with computer-assisted specification, automatic model generation, automatic model management, and knowledge-based simulation analysis. The implementation of these environments will dramatically change the manner in which design is conducted.

Future research into these environments will include such topics as:

1. the implementation of GIBSS on parallel architectures,
2. the applicability of object oriented programming for GIBSS modeling, and
3. automatic model update,
4. automatic data collection, and
5. automatic model generation.

BIBLIOGRAPHY

1. Almasi, G. S., and A. Gottlieb, Highly Parallel Computing, Benjamin/Cummings Publishing, Inc., Reading, MA, 1989.

2. Antonelli, C. J., Volz, R. A., and T. N. Mudge, "Hierarchical decomposition and simulation of manufacturing cells using Ada," Simulation, April 1986, pp. 141-152.

3. Appleton D., Information Modeling Manual: IDEF 1 Extended, D. Appleton Comp., December, 1985.

4. Balci, O., "The Implementation of Four Conceptual Frame Works for Simulation Modeling in High Level Languages," Proceedings of 1988 Winter Simulation Conference, San Diego, CA.

5. Balci, O. and R. E. Nance, "Simulation Support: Prototyping the Automation-Based Paradigm," Proceedings of the 1987 Winter Simulation Conference, Atlanta, GA, pp. 535-550.

6. Balci, O. and R. E. Nance, "Formulated Problem Verification as an Explicit Requirement of Model Credibility," Simulation, August 1985, pp. 76-86.

7. Balzer, R. "A 15 Year Perspective on Automatic Programming,"IEEE Trans. on Software Engineering, Vol. SE-11, No. 11, pp. 1257-1269.

8. Berry, D. T., "Stamping Out Forming Problems with FEA," Mechanical Engineering, July 1988, pp. 58-62.

9. Boling, Beth and Marc Layman, "ModelMaster Factory Modeling System Tutorial," 1986 Winter Simulation Conference Proceedings, Washington D.C., pp. 156-159.

10. Chace, A., "Modeling of dynamic mechanical systems," Conference on CAD/CAM Robotics and Automation, Tucson, Feb. 1985.

11. Concepcion, A. and S. Schon, "SAM - A Computer Aided Design Tool for Specifying and Analyzing Modular, Hierarchical Systems," Proceedings of the 1986 Winter Simulation Conference, Washington D.C., pp. 504-510.

12. Concepcion, A. and B. P. Zeigler, "DEVS Formalism: A Framework for Hierarchical Model Development," IEEE Trans. Sotfware Engineering, 1987.

13. Conway, Richard and William Maxwell, "Modeling Asynchronous Materials Handling Systems in XCELL," 1987 Winter Simulation Conference Proceedings, Atlanta, GA, pp. 202-206.

14. Dubowsky, S. and R. Kornbluh, "On the development of high performance adaptive control algorithms for robotics," Robotics Research 2nd International Symposium, Cambridge, MA, 1985, pp. 119-126.

15. Engelke, H., Grotrian, J., and A. Schmackpheffer, "Manufacturing Description Method," Proceedings of the 1987 Summer Simulation Conference, Montreal, Quebec, pp. 625-630.

16. Farnsworth, K. D., Norman, V. B., and T. A. Norman, "Integrated Software for Manufacturing Simulation," 1987 Winter Simulation Conference Proceedings, Atlanta, GA, pp. 195-201.

17. Fernandez, K. and E. Hinman, "The Use of Computer Graphic Simulation in the Development of On-Orbit Tele-Robotic Systems," ROBOTS 11 Conference Proceedings, Society of Manufacturing Engineers, Chicago, 1987, pp. 21-45-21-54.

18. Fletcher, R. W. and A. A. Goldenberg, "Collision Avoidance for Robot Manipulators: Application to CATIA/IBM 7565 Interface," Journal of Robotic Systems, Sept. 1988, pp. 125-146.

19. Ford, D. R., Schroer, B. J. and R. Daughtrey, "An Intelligent Modeling System for Simulating Manufacturing Processes," 1987 Winter Simulation Conference Proceedings, Atlanta, GA, pp. 525-529.

20. Fougere, T. and J. Kanerva, "Robot-Sim - A CAD Based Workcell Design and Off-Line Programming System," ROBOTS 10 Conference Proceedings, Society of Manufacturing Engineers, Detroit, MI

21. Fleury, C., "Future Trends in Computer Aided Optimal Design," in Computer Aided Optimal Design: Structural and Mechanical Systems, C. A. Soares ed., Springer-Verlag, New York, 1987.

22. Gilman, A. and R. M. Watremez, "A Tutorial on SEE WHY and WITNESS," 1986 Winter Simulation Conference Proceedings, Washington D.C., pp. 178-183.

23. Gilmore, B., The simulation of mechanical systems with a changing topology, Ph.D. dissertation, Dept. Mech. Eng., Purdue, Lafayette, IN, August. 1986.

24. Glassner, A. and H. Fuchs, "Hardware Enhancements for Raster Graphics," in Fundamental Algorithms for Computer Graphics, R. A. Earnshaw ed., Springer-Verlag, New York, 1985.

25. Haddock, J., "A Simulation Generator for Flexible Manufacturing Systems Design and Control," IIE Transactions, Vol. 20, No. 1, 1988, pp. 22-30.

26. Hoffman, C. M. and J. E. Hopcroft, "Simulation of Physical Systems from Geometric Models," IEEE Journal of Robotics and Automation, June 1987, pp. 194-206.

27. Imam, I., Davis, J., and T. Fougere, "Flexible Manufacturing Cell Simulation," ROBOTS 11 Conference Proceedings, Society of Manufacturing Engineers, Chicago, 1987, pp. 21-55 - 21-74.

28. Jones, C., "Applications of Offline Programming," ROBOTS 11 Conference Proceedings, Society of Manufacturing Engineers, Chicago, Il, pp. 21.25- 21.43.

29. Lenz, J. E., "MAST: a simulation tool for designing computerized metalworking factories," Simulation, Feb. 1983, pp. 51-58.

30. Mackulak, G., "High Level Planning and Control: An IDEF Analysis for Airframe Manufacture," Journal of Manufacturing Systems, Vol. 3, No. 2., 1984, pp. 121-134.

31. Martin, D., Advanced Data Base Techniques, MIT PRESS, Cambridge, MA, 1986.

32. Medeiros, D. J. and R. P. Sadowski, "Simulation of robotic manufacturing cells: a modular approach," Simulation, January 1983, pp. 3-12.

33. McFarland, D. G., "Scheduling Manufacturing Systems with FACTOR," 1987 Winter Simulation Conference Proceedings, Atlanta, GA, pp. 235-237.

34. Mogul, J. S., "IGRIP - A Graphics Simulation Program for Workcell Layout and Off-Line Programming," ROBOTS 10 Conference Proceedings, Society of Manufacturing Engineers, Detroit, MI, 1986.

35. Mathewson, S. C., "The Application of Program Generator Software and its Extensions to Discrete Event Simulation Modeling," IIE Transactions, Vol. 16, No. 1, March 1984.

36. Mortenson, M. E., Geometric Modeling, John Wily, New York, 1985.

37. Nikravesh, P. E., "Applications of Animated Graphics for Large Scale Mechanical System Dynamics," in Computer Aided Analysis and Optimization of Mechanical Systems, E. J. Haug ed., Springer-Verlag, New York, 1984.

38. O'Keefe, R. M., "What is Visual Interactive Simulation?," Proceedings of the 1987 Winter Simulation Conference, Atlanta, GA, pp. 461-464.

39. Pascoe, G. A., "Elements of Object Oriented Programming," BYTE, August 1986, pp. 139-144.

40. Paul, R., and H. Zhang, "Robot Motion trajectory specification and generation, Robtics Research, 2nd International Symposium, Cambridge, MA, 1985, pp. 187-193.

41. Pegden, C. D., Introduction to SIMAN, Systems Modeling, Penn., Corp., 1982.

42. Pritsker, A and B. Alan, Introduction to Simulation and SLAM II, Halsted Press, New York, 1985.

43. Rebelo, N., "FEA of Forming," Machine Design, June 9, 1988.

44. Shannon, Robert E., Systems Simulation: The Art and Science, New York, Prentice-Hall, 1975.

45. Speed, R., "Offline Programming for Industrial Robots with CAD/CAM Assistance," ROBOTS 11 Conference Proceedings, Chicago, Il, pp. 21.1-21.24.

46. Steudel, H. J. and T. Park, "STAR*CELL: A Flexible Manufacturing Cell Simulator," 1987 Winter Simulation Conference Proceedings, pp. 230-234.

47. Surl, R., "RMT Puts Manufacturing at the Helm," Industrial Engineering, Feb. 1988, pp. 41-44.

48. Tumay, K., "Factory Simulation with Animation: The No Programming Approach," 1987 Winter Simulation Conference Proceedings, Atlanta, GA, pp. 258-260.

49. U.S.A.F, Function Modeling Manual (IDEF 0), Wright-Patterson Air Force Base, Ohio, June 1981.

50. U.S.A.F, Dynamics Modeling Manual (IDEF 2), Wright-Patterson Air Force Base, Ohio, June 1981.

51. Wang, W. P., "Solid Modeling for Optimizing Metal Removal of Three-dimensional NC End Milling," Journal of Manufacturing Systems, Vol. 7, No. 1, pp. 57-65.

52. White, D., "PCModel and PCModel/GAF - Screen Oriented Modeling," 1986 Winter Simulation Conference Proceedings, Washington D.C., pp. 164-167.

53. Wysk, R. A., Cohen, P. H., Ghosh, B. K., and S. Y. Wu, "Advantages of Scaled and Unscaled Physical Models for FMS Research and Instruction," Journal of Manufacturing Systems, Vol. 6, No. 2., pp. 107-115.

54. Zeigler, B. P., "Theory and Application of Modeling and Simulation: A Sotware Engineering Perspective," in Handbook of Software Engineering, C. R. Vick ed., Van Nostrand Reinhold, New York, 1984.

55. Zeigler, B. P., "Hierarchical Modular Modeling/Knowledge Representation," Proceedings of the 1986 Winter Simulation Conference, Washington D.C., pp. 129-137.

56. Zeigler, B. P., "Hierarchical, modular discrete-event modeling in an object oriented environment," Simulation, Nov. 1987, pp. 219-230.

57. Zeigler, B. P. and T. I. Oren, "Concepts for advanced simulation methodologies," Simulation, March 1979, pp. 69-82.

APPENDIX A WORK CELL DESIGN

A.0 Introduction

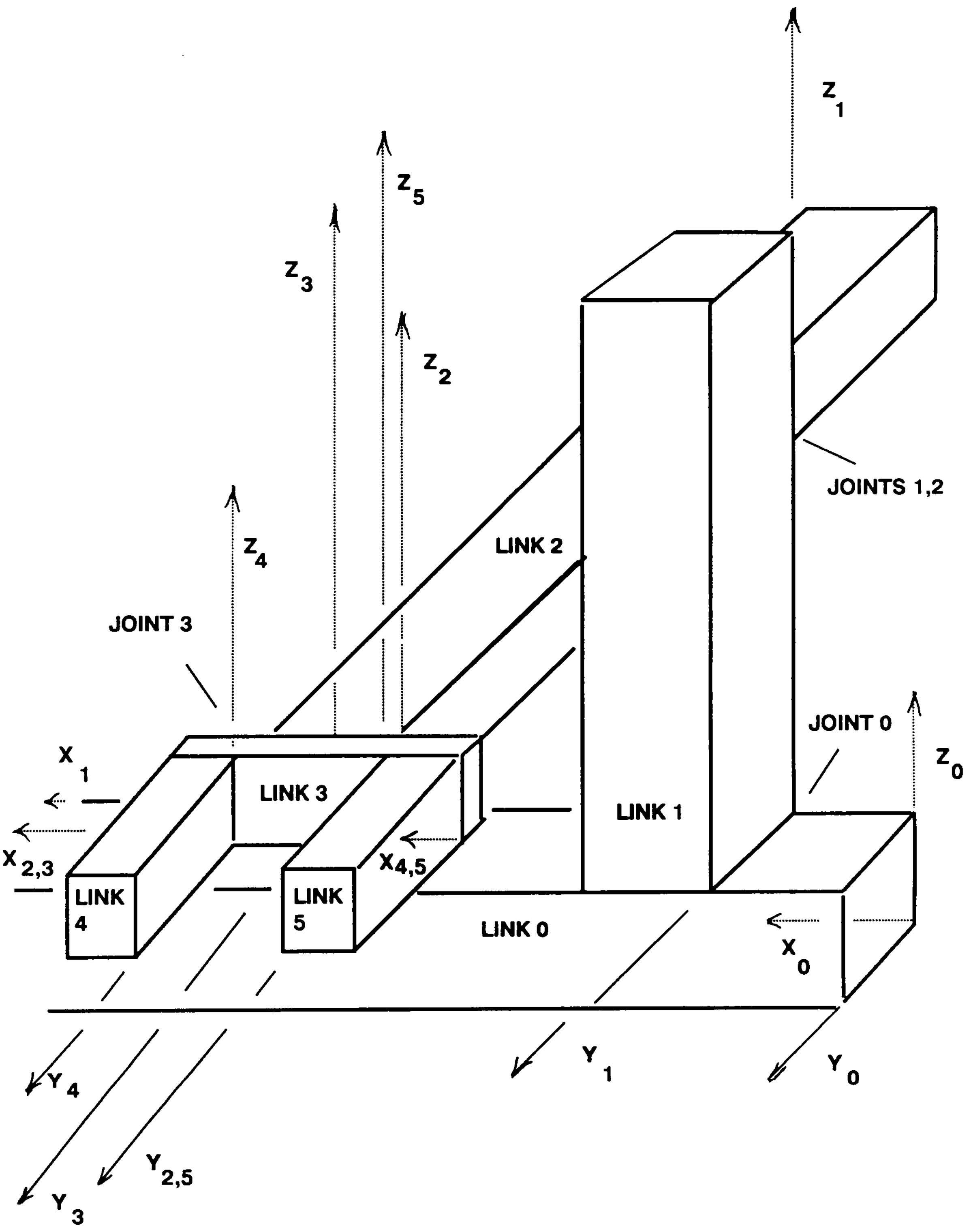
This appendix describes the geometry, kinematics, and control of the hypothetical manufacturing system. Section A.1 provides isometric drawings and part drawings of the following work cell classes:

1. robot,
2. machine tool,
3. pallet, and
4. part.

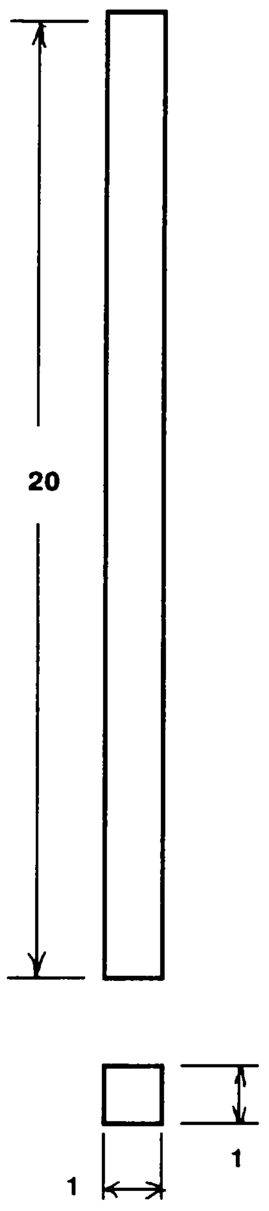
In addition, the links and joints associated with the robot and machine tool classes are identified along with their frames. Section A.2 describes the kinematics of the robot and machine tool classes. Section A.3 describes the control program structure of the robot and machine tool classes and lists the programs of the robots and machine tools.

A.1 Manufacturing System Geometry

This section provides isometric drawings and part drawings of the robot, machine tool, pallet, and part classes. Each isometric drawing illustrates the reference frame(s) for its respective class. In addition, the joints and links for the robot and machine tool classes are identified. At the end of this section, the base frame coordinates for the robots, machine tools, and pallets are provided. These coordinates are relative to a cartesian coordinate system established for the hypothetical manufacturing system.



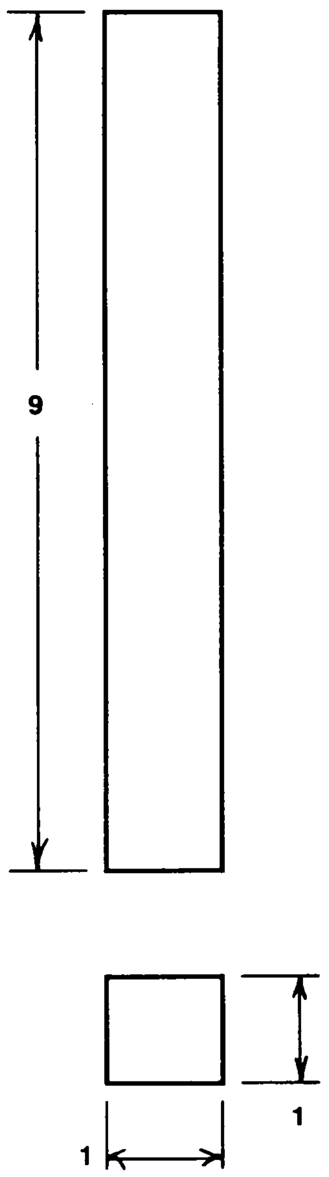
Isometric Sketch of Robot Class



UNITS = feet

SCALE = 1:48

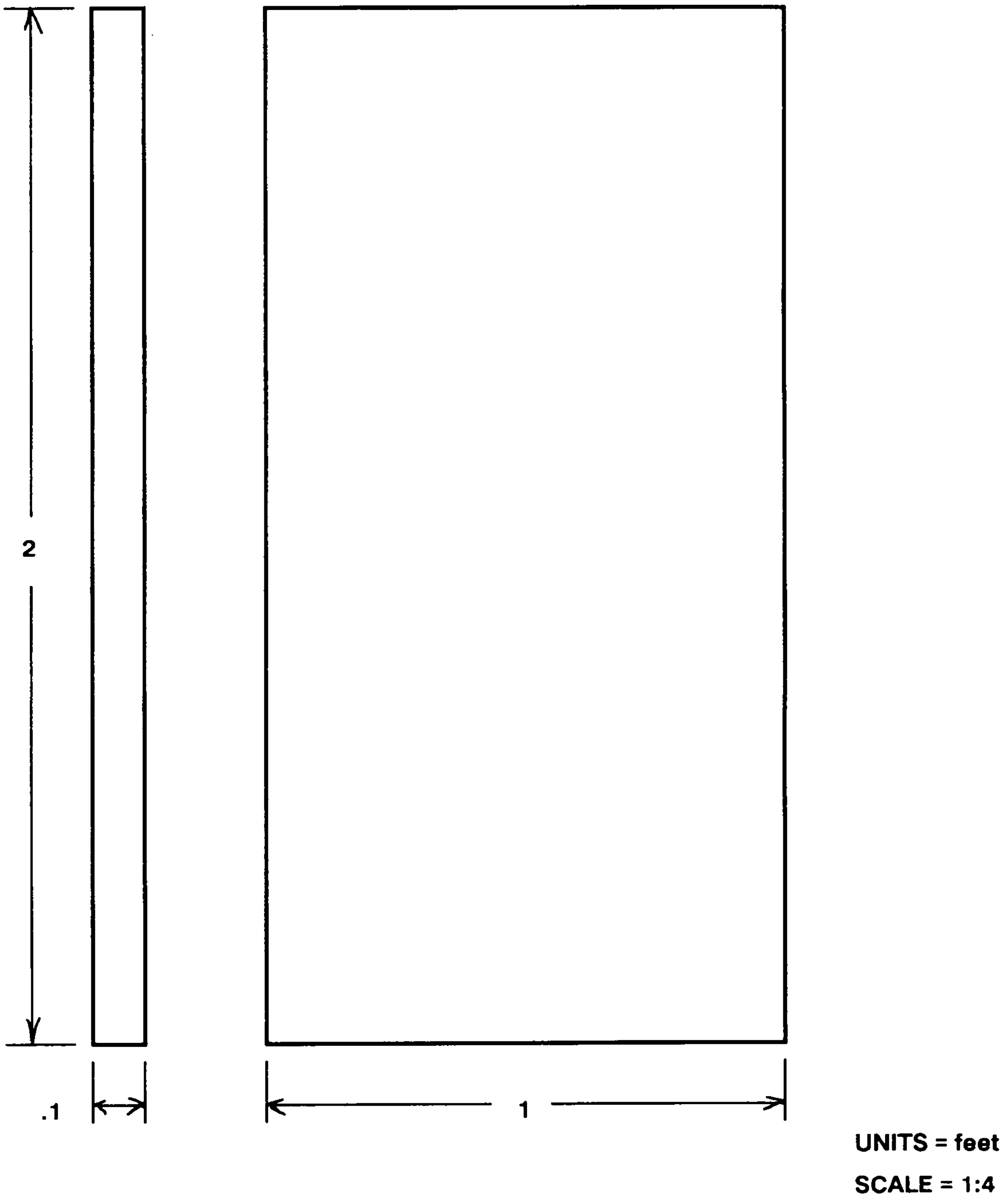
Drawing of Robot Class Link 0



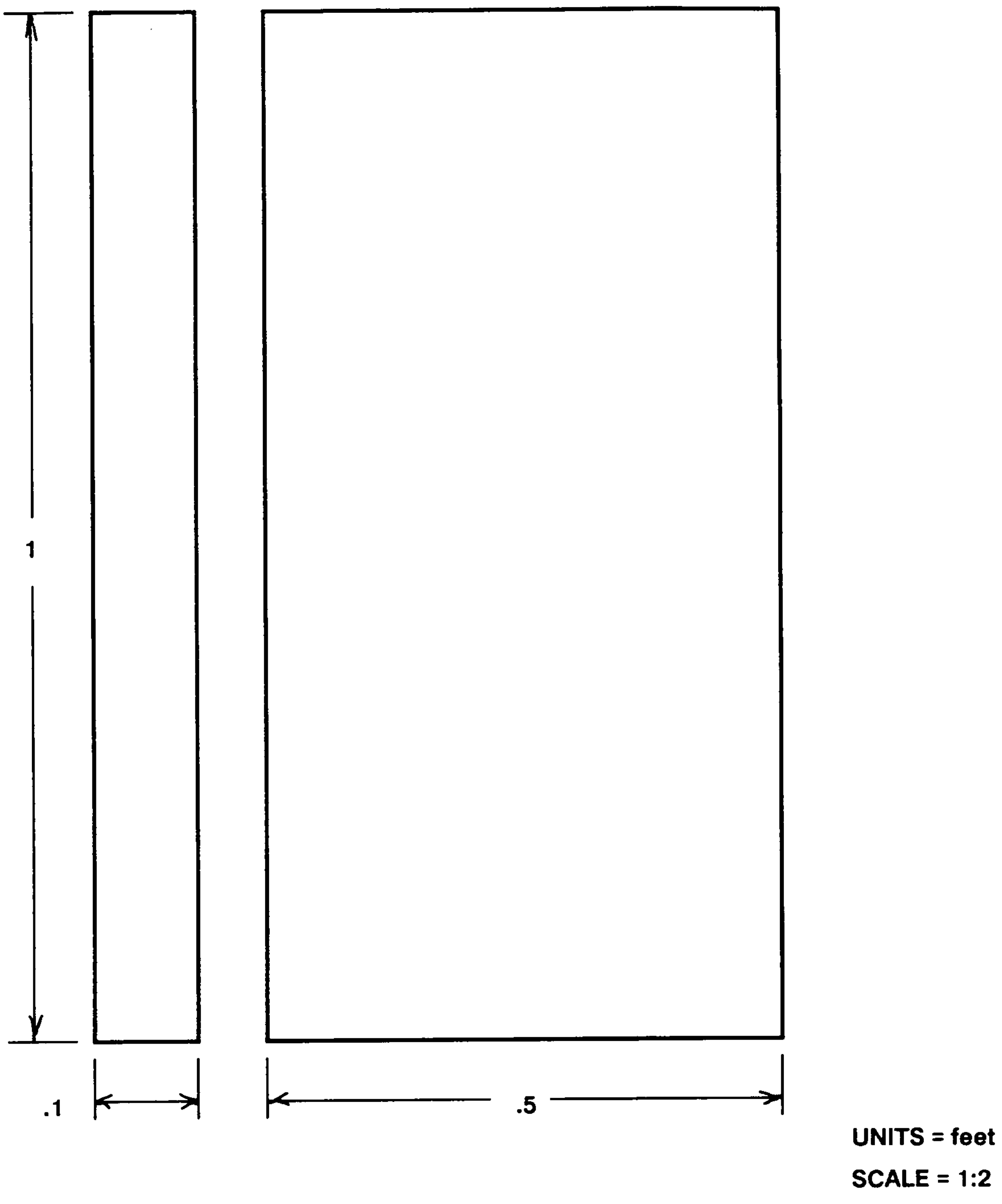
UNITS = feet

SCALE = 1:24

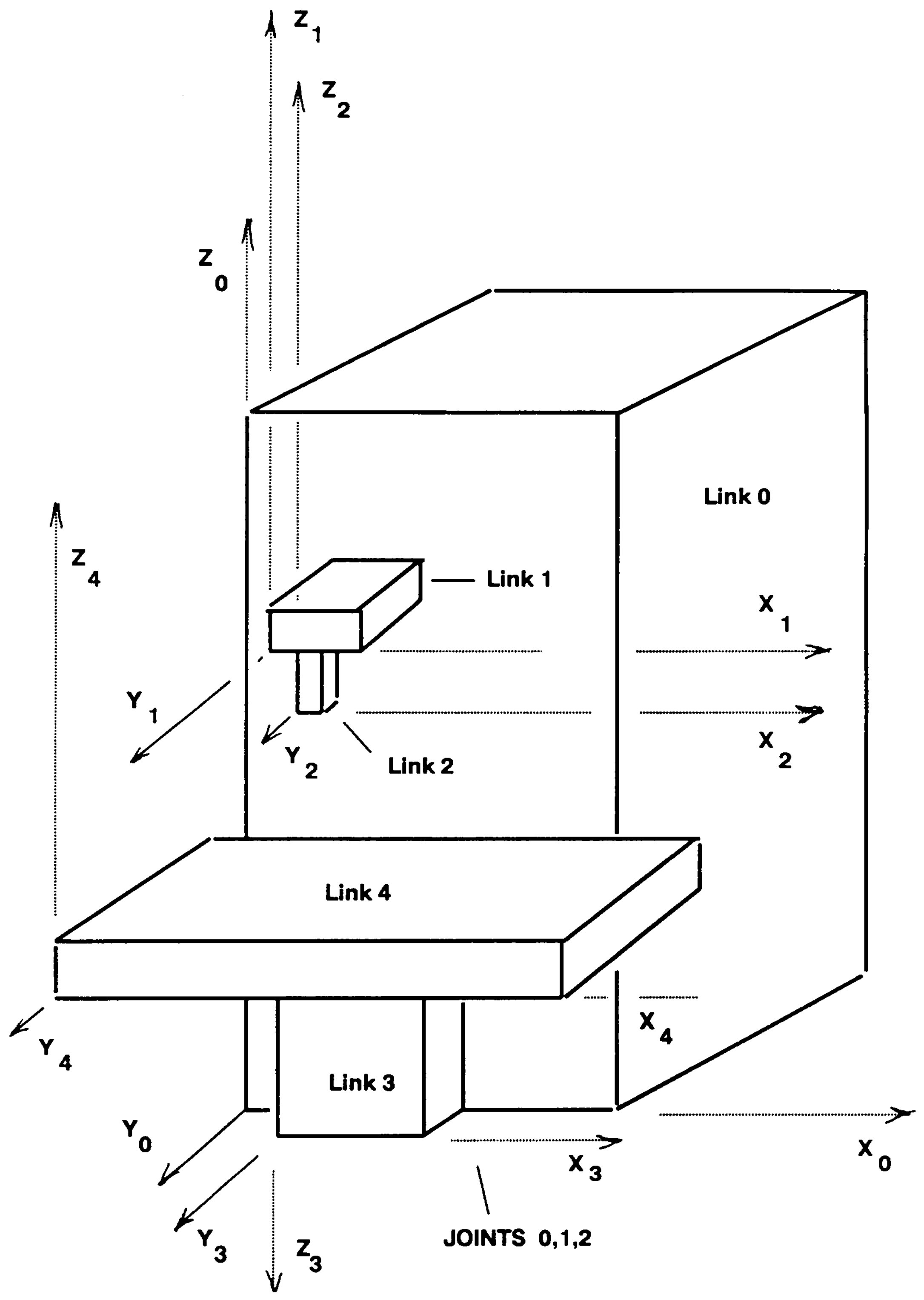
Drawing of Robot Class Links 1 and 2



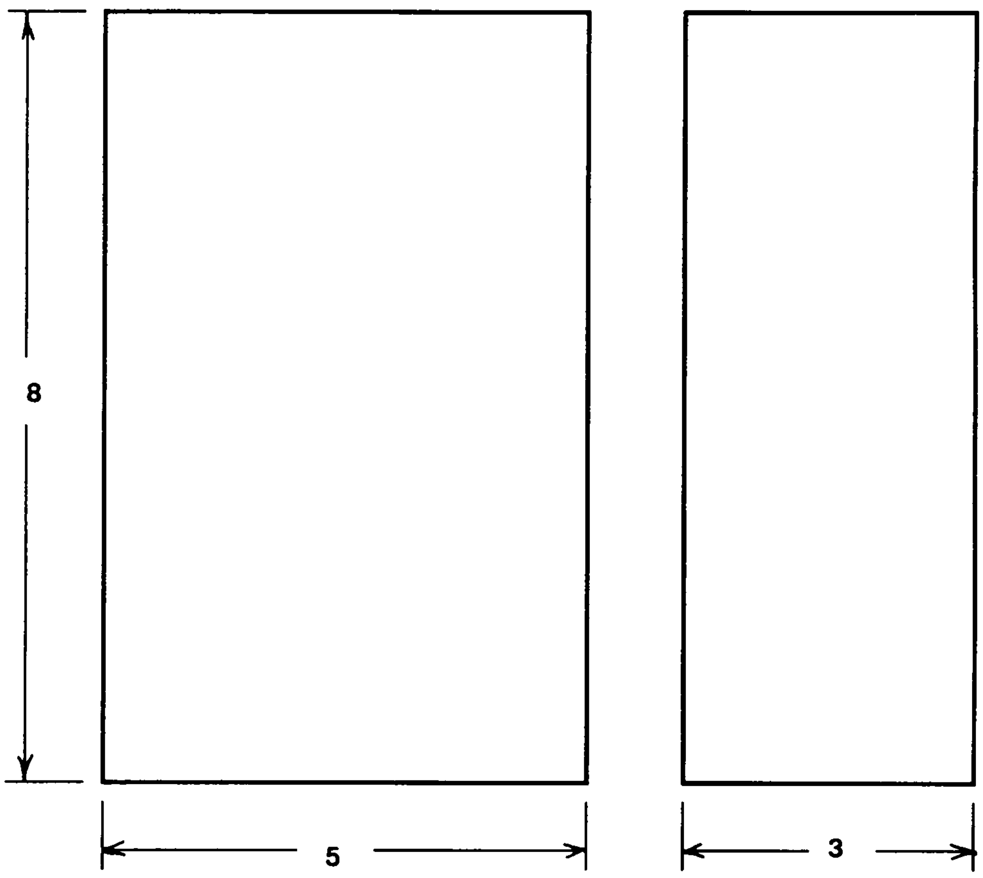
Drawing of Robot Class Link 3



Drawing of Robot Class Links 4 and 5



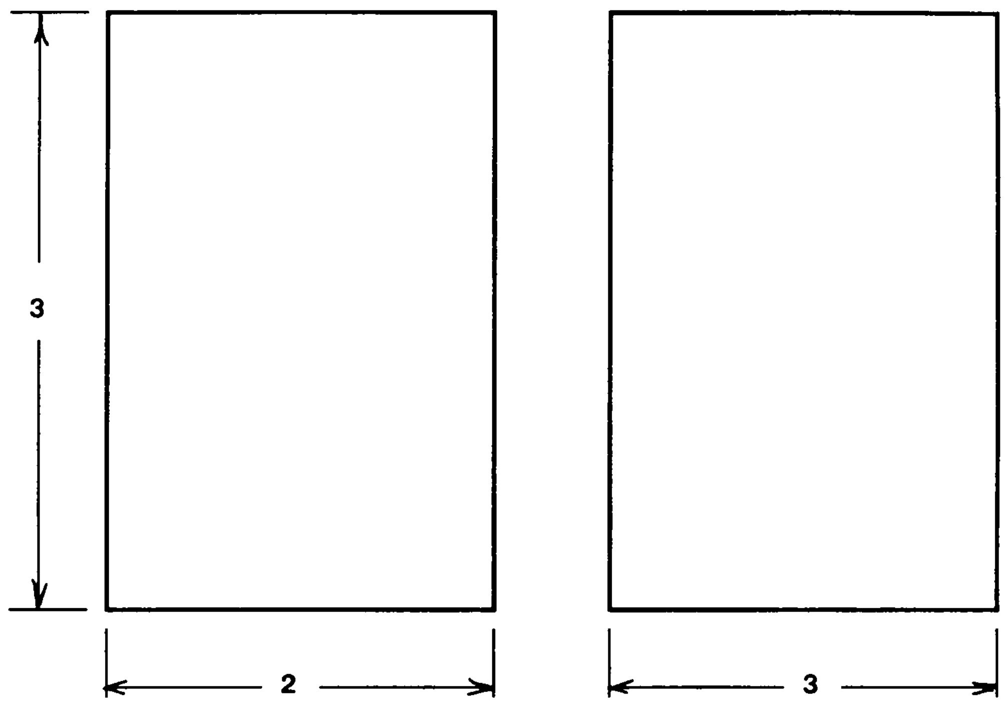
Isometric Sketch of Machine Tool Class



UNITS = feet

SCALE = 1:24

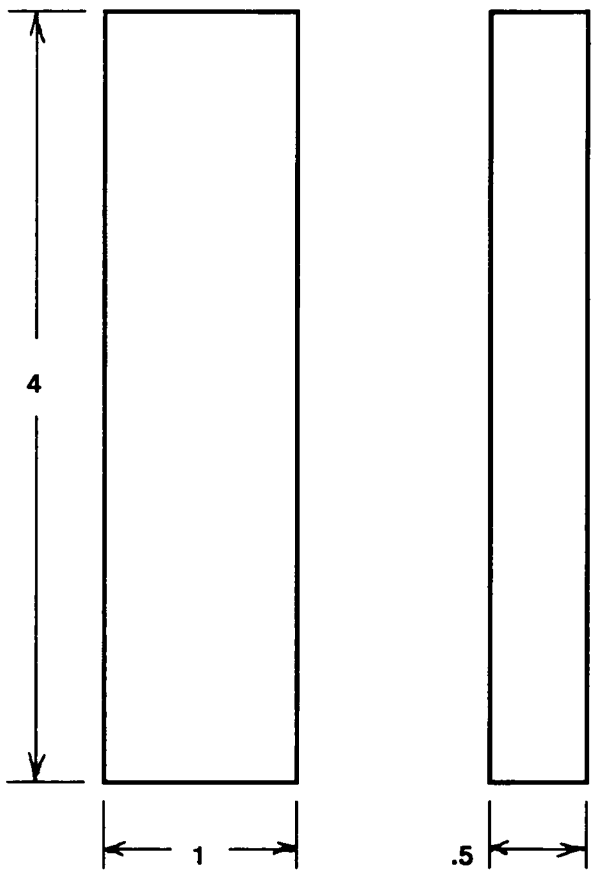
Drawing of Machine Tool Class Link 0



UNITS = feet

SCALE = 1:12

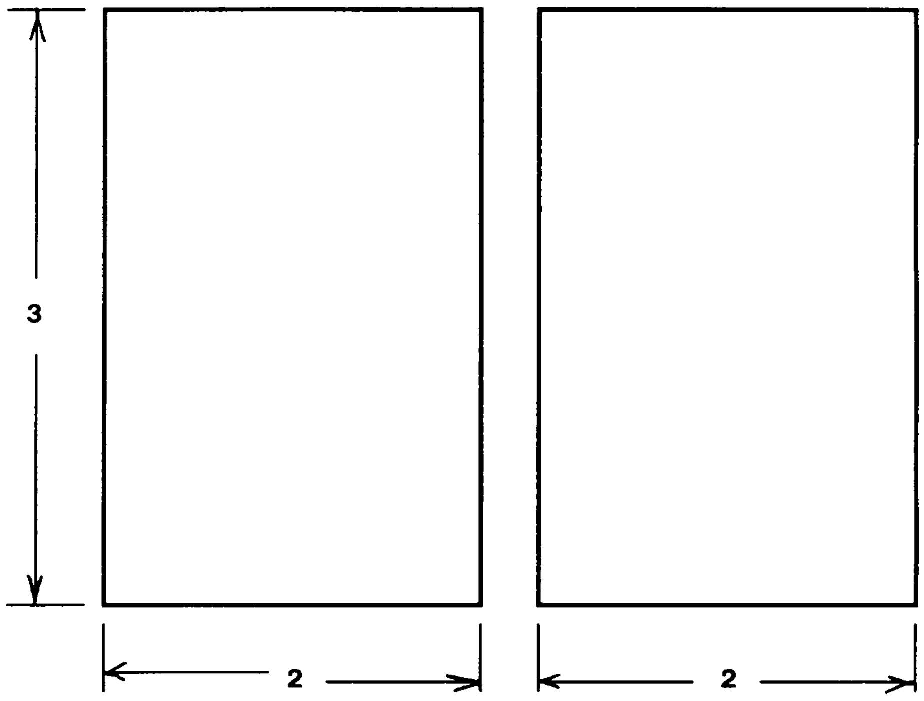
Drawing of Machine Tool Class Link 3



UNITS = feet

SCALE = 1:12

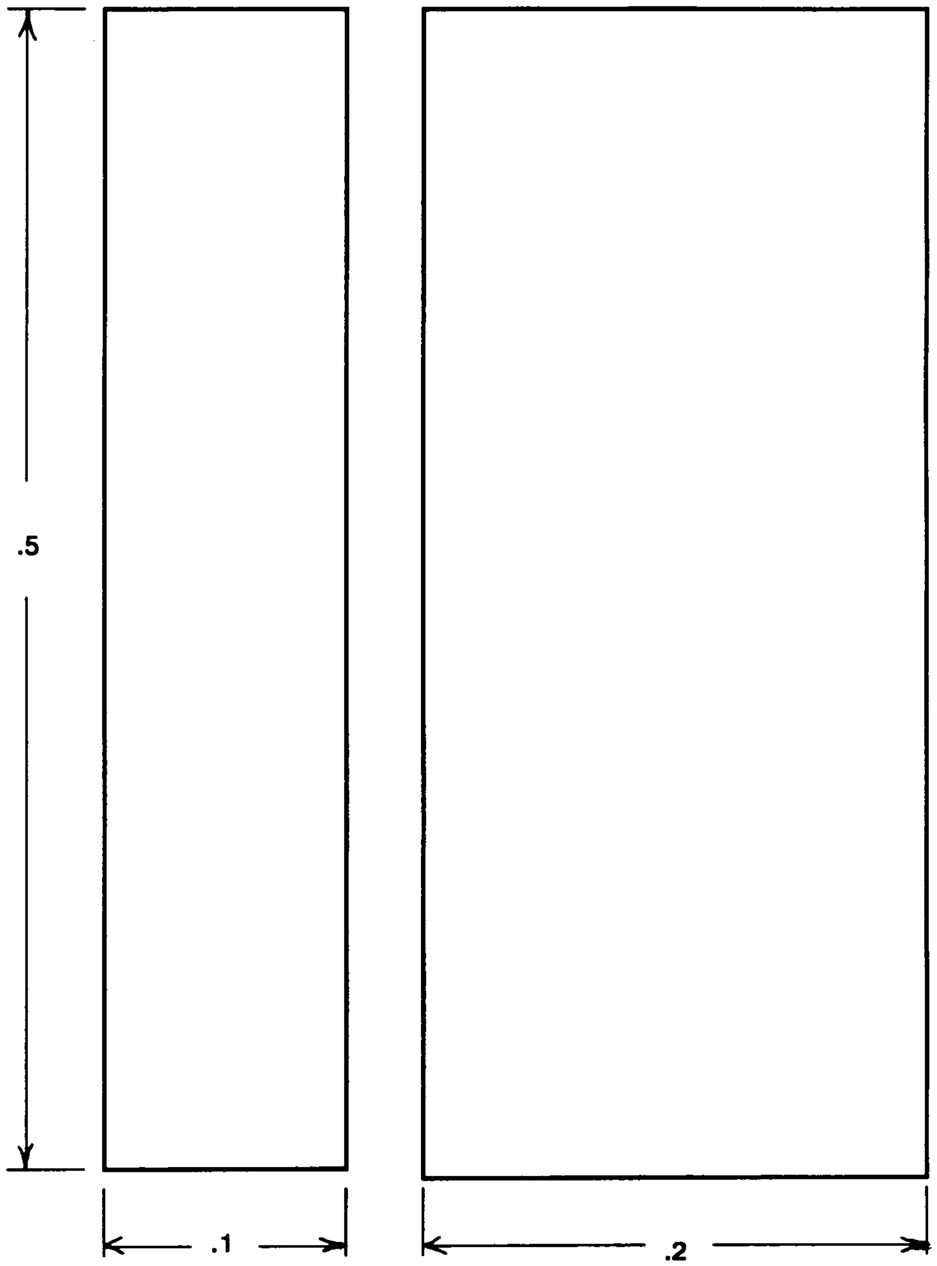
Drawing of Machine Tool Class Link 4



UNITS = feet

SCALE = 1:12

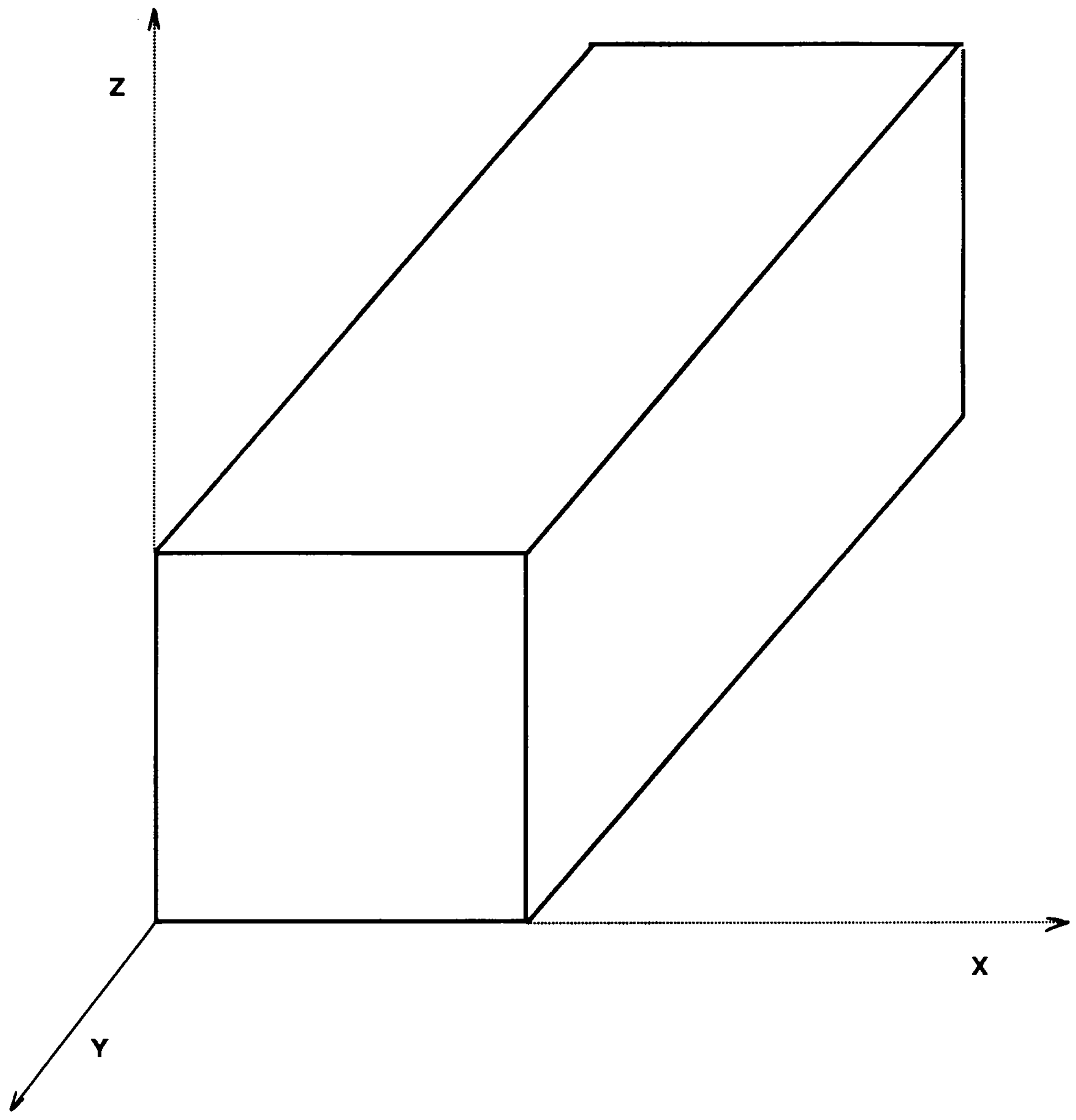
Drawing of Machine Tool Class Link 1



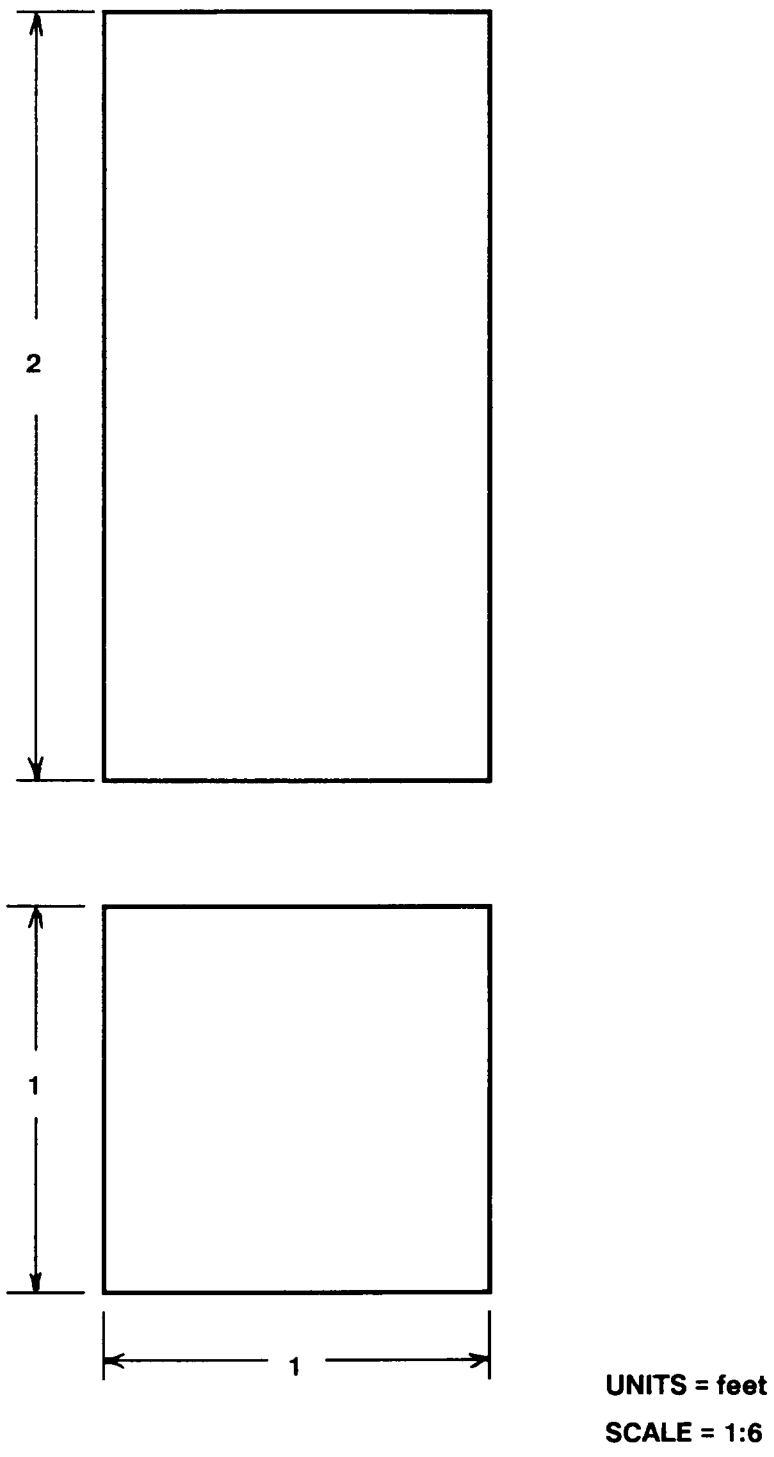
UNITS = feet

SCALE = 1:1

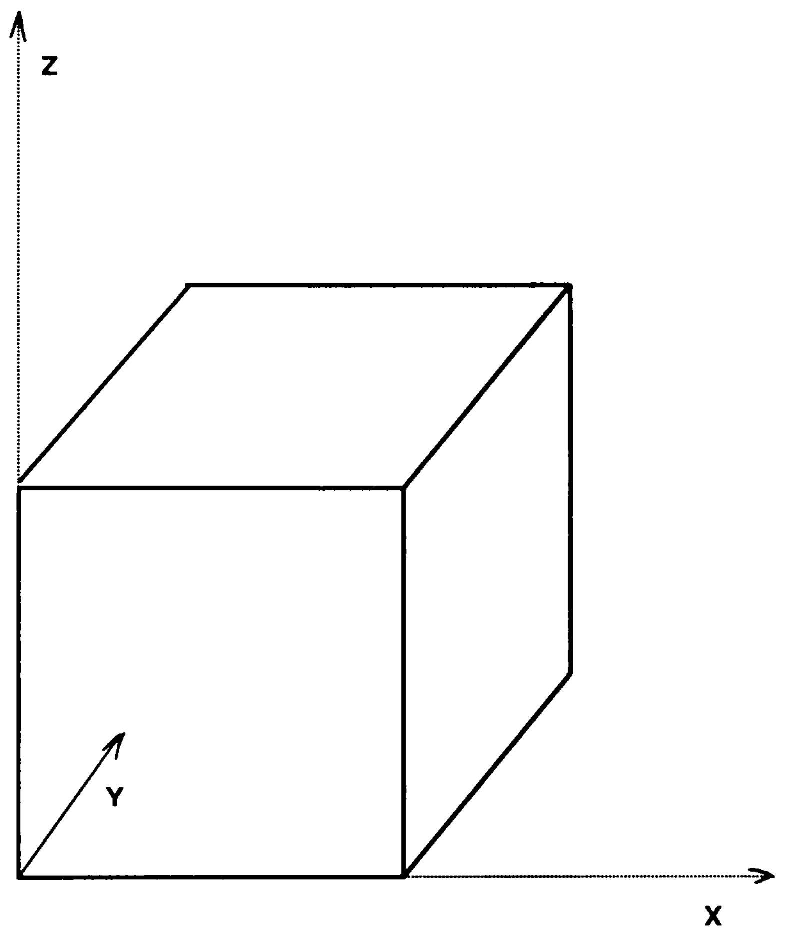
Drawing of Machine Tool Class Link 2 (Cutting Tool)



Isometric Sketch of Pallet Class

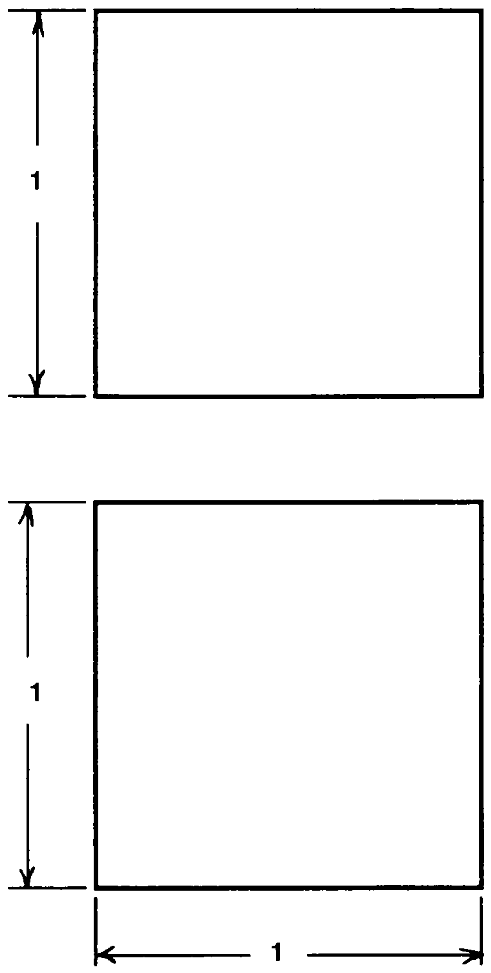


Drawing of Pallet Class



No. of Points = 8
No. of Edges = 12
No. of Faces = 6
Area = 6
Volume = 1

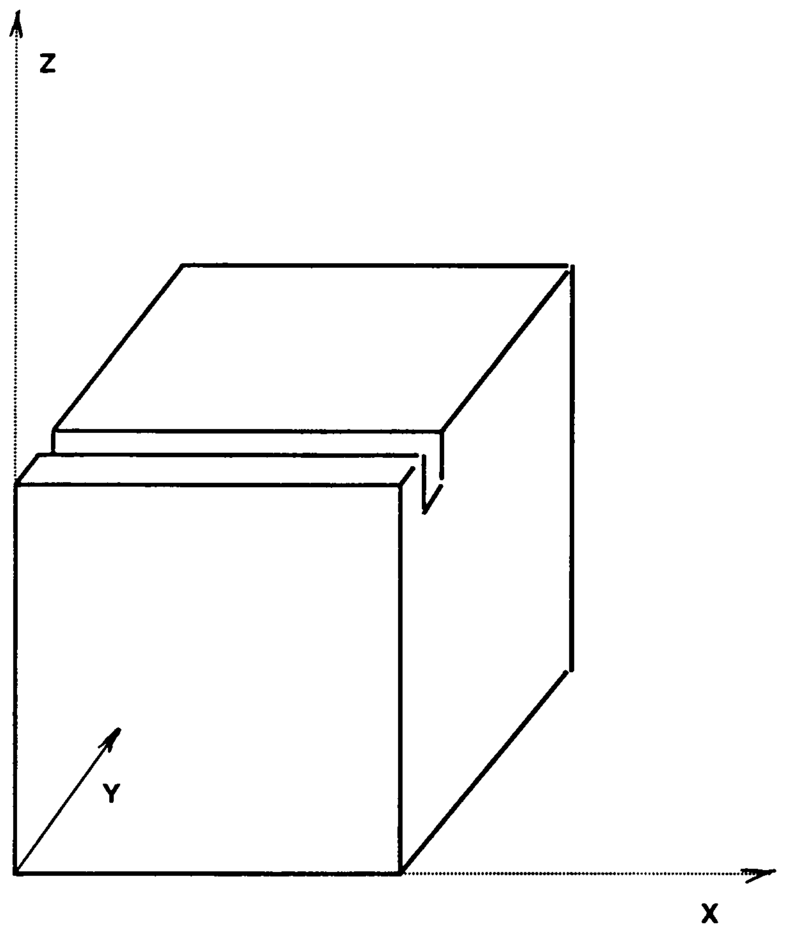
Isometric Sketch of Part Class 0



UNITS = feet

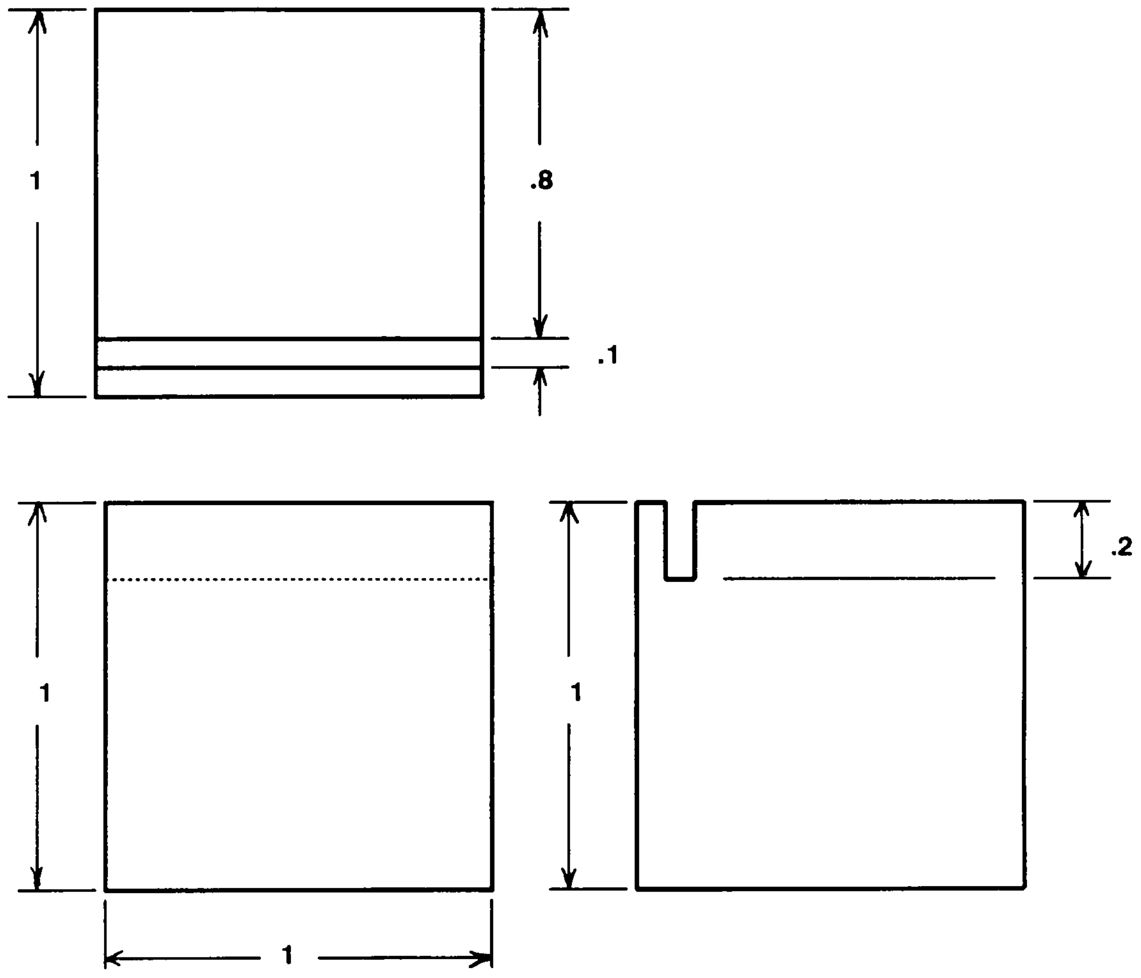
SCALE = 1:6

Drawing of Part Class 0



No. of Points = 20
No. of Edges = 30
No. of Faces = 10
Area = 6.36
Volume = .98

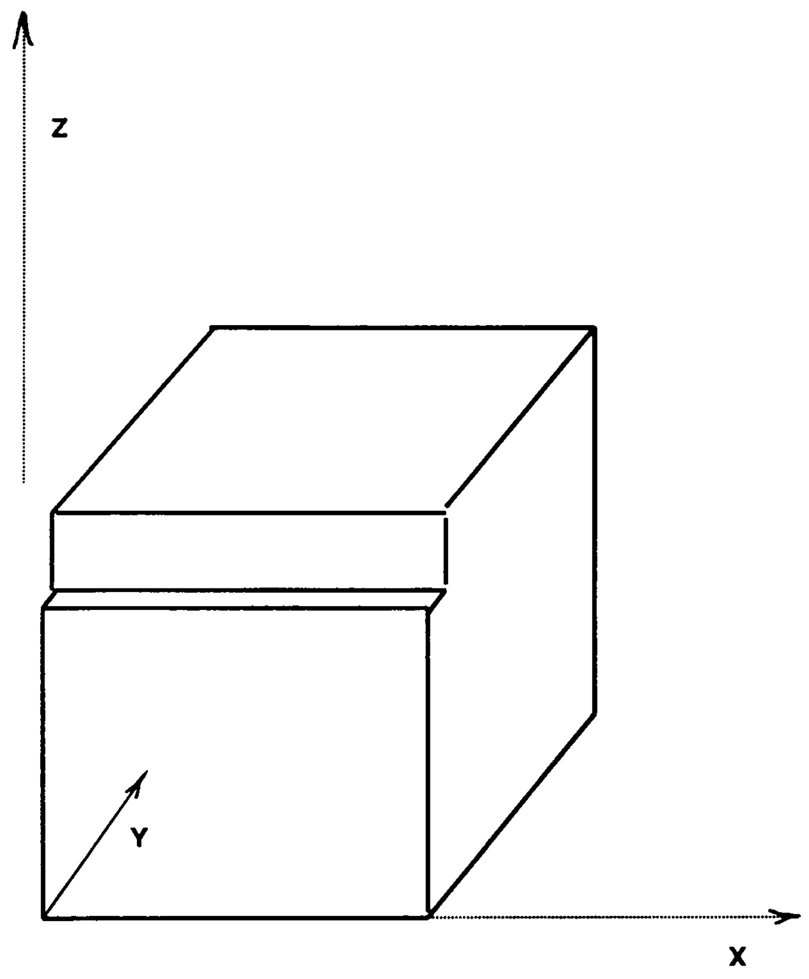
Isometric Sketch of Part Class 1



UNITS = feet

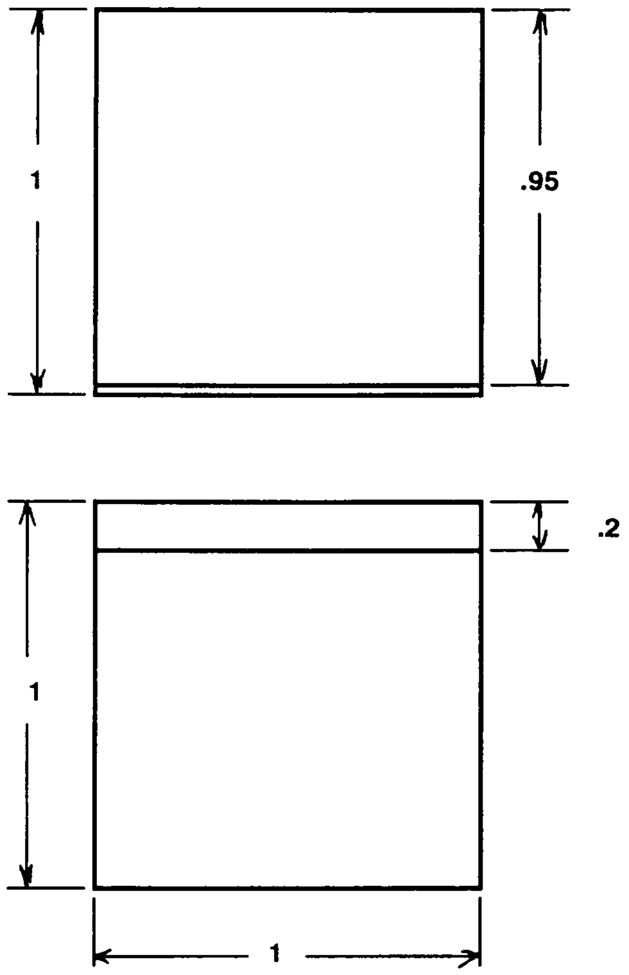
SCALE = 1:6

Drawing of Part Class 1



No. of Points = 12
No. of Edges = 18
No. of Faces = 8
Area = 5.98
Volume = .99

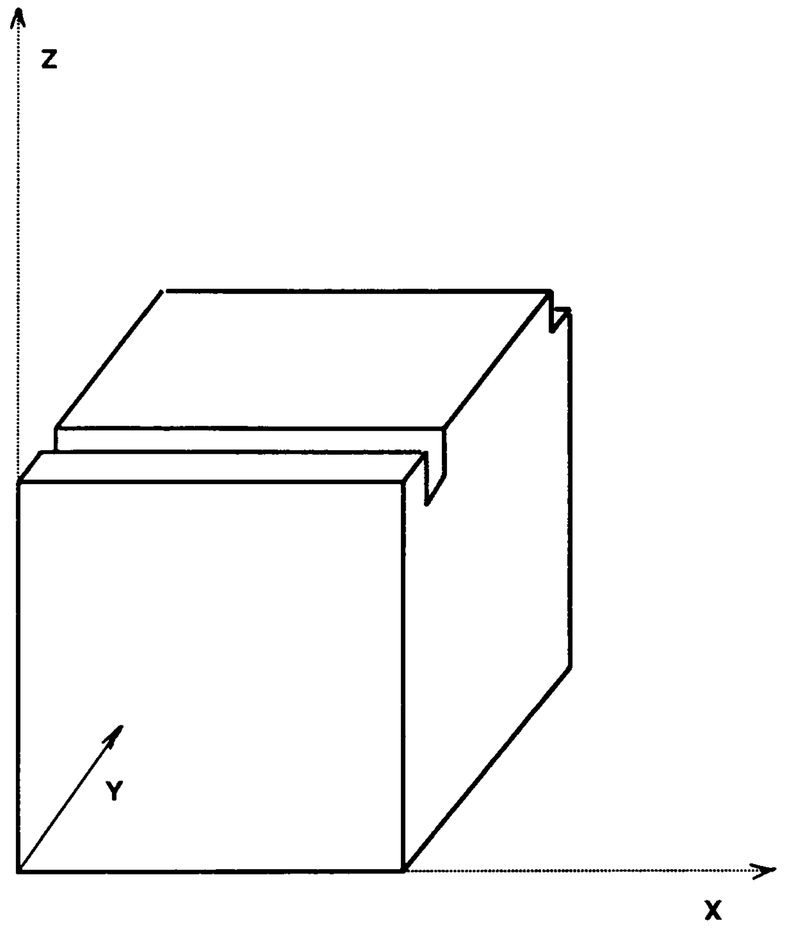
Isometric Sketch of Part Class 2



UNITS = feet

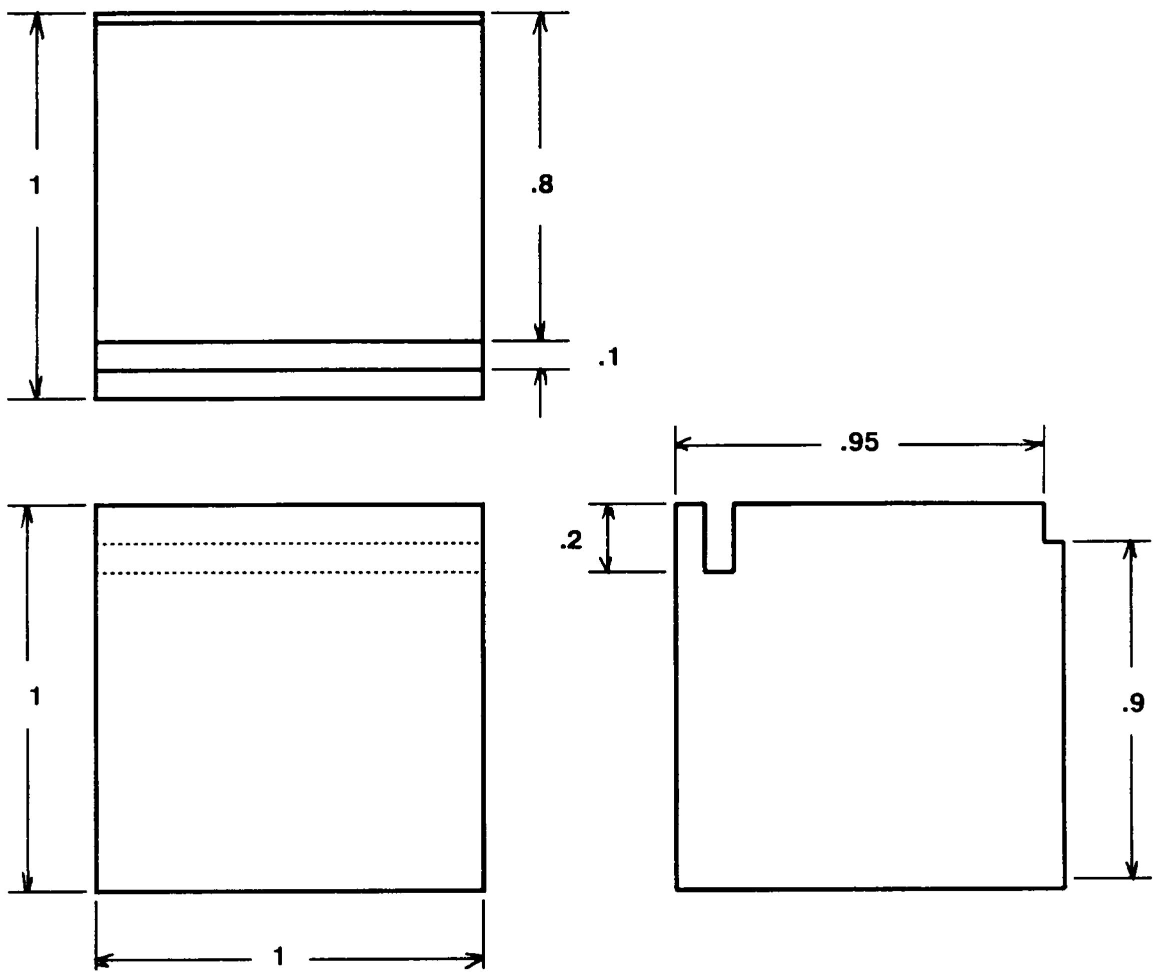
SCALE = 1:6

Drawing of Part Class 2



No. of Points = 20
No. of Edges = 30
No. of Faces = 12
Area = 6.26
Volume = .975

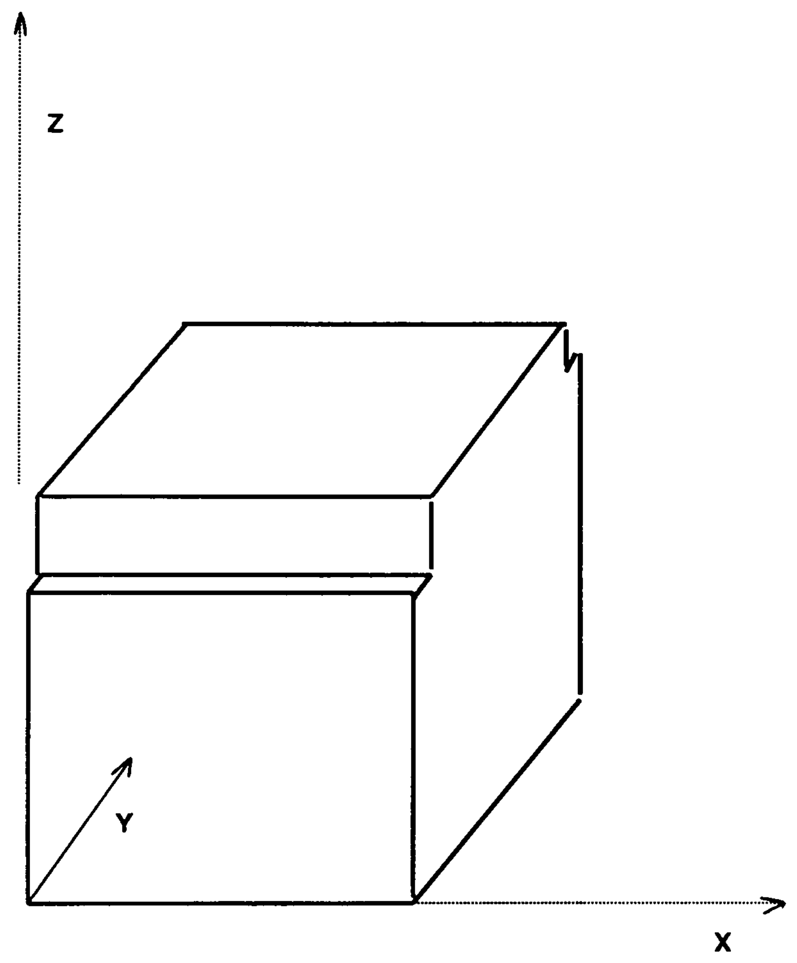
Isometric Sketch of Part Class 3



UNITS = feet

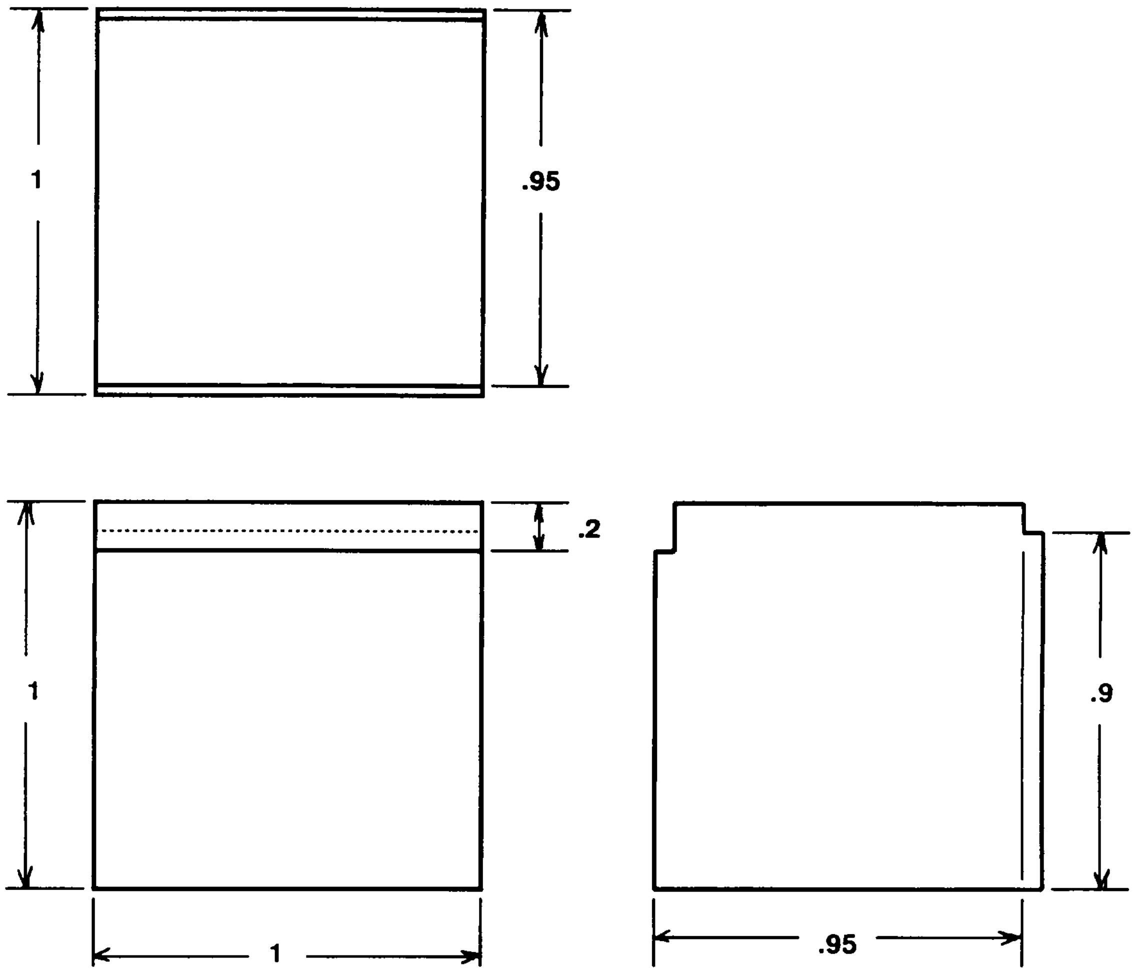
SCALE = 1:6

Drawing of Part Class 3



No. of Points = 16
No. of Edges = 24
No. of Faces = 10
Area = 5.97
Volume = .985

Isometric Sketch of Part Class 4



UNITS = feet

SCALE = 1:6

Drawing of Part Class 4

Coordinates of Work Cell Resources

	Resource	X	Y	Z
	Pallet 1	3	3	0
*	Robot 1	0	0	0
*	Machine Tool 1	9.25	6	0
	Pallet 2	19.5	3	0
*	Robot 2	20	0	0
*	Machine Tool 2	25.25	6	0
	Pallet 3	36	3	0

* With respect to LINK 0

A.2 Manufacturing System Kinematics

All motion within the hypothetical manufacturing system is linear. Kinematically, a body is represented with a position vector, **P**, and a velocity vector , **V**, which represent the position and linear velocity of the body with respect to the general coordinate system.

Each manipulator has a vector, **J**, which represents the linear displacements of its joints, and a vector, $\mathbf{J_v}$, which represents the linear velocities of its joints. Manipulator link position and velocity are functions of **J** and $\mathbf{J_v}$ respectively. In general, link position is computed by:

$$\mathbf{P} = \mathbf{TJ} + \mathbf{O} + \mathbf{P_0} \quad (1)$$

where **T** = joint transformation matrix,

O = link offset vector, and

$\mathbf{P_0}$ = position vector of LINK 0.

For a given link, multiplication of **T** and **J** and the addition of **O** computes its position with respect to the LINK 0 reference frame. The LINK 0 reference frame is stationary and dictates the position of a manipulator with respect to the general coordinate system. The addition of $\mathbf{P_0}$ to the sum computes the position of the link within the general coordinate system.

Link velocity is computed by:

$$\mathbf{V} = \mathbf{T}\mathbf{J}_{\mathbf{V}} \quad (2)$$

The values of **T** and **O** for links of the robot and machine tool classes are provided in the following tables.

Kinematic Parameters for Robot Class

LINK 0

T	O
0 0 0 0	0 0 0
0 0 0 0	
0 0 0 0	

LINK 1

T	O
1 0 0 0	0 0 1
0 0 0 0	
0 0 0 0	

LINK 2

T	O
1 0 0 0	1 1 1
0 1 0 0	
0 0 1 0	

LINK 3

T	O
1 0 0 0	1.5 1 1
0 1 0 0	
0 0 1 0	

LINK 4

T	O
1 0 0 .5	1.5 1.1 1
0 1 0 0	
0 0 1 0	

LINK 5

T	O
1 0 0 -.5	1.5 1.1 1
0 1 0 0	
0 0 1 0	

Kinematic Parameters for Machine Tool Class

LINK 0

T	O
0 0 0	0 0 0
0 0 0	
0 0 0	

LINK 1

T	O
1 0 0	2 -2 6
0 0 0	
0 0 0	

LINK 2

T	O
0 0 0	2.45 -1 5.5
0 0 0	
0 0 0	

LINK 3

T	O
0 0 0	1.5 -3 0
0 0 0	
0 0 1	

LINK 4

T	O
1 0 0	-.5 -3 2
0 1 0	
0 0 1	

A.3 Manufacturing System Control

The robots and machine tools within the hypothetical manufacturing system control their actions through the scanning and execution of text programs. The programs consist of numeric instructions structured into N X 6 arrays, where N represents the number of program instructions. The first column in the array is used for instruction identification. Eight different types of instructions exist, each with their own particular format. These instruction types are **PMOVE, DELAY, WAITI, TESTI, WRITEO, JUMP, TESTC,** and **SETC.**

A PMOVE statement is indicated by a "1." It specifies the joint movements of a manipulator over a segment of time. Trajectory is specified by parameters stored in the subsequent five columns of the instruction. Columns 2 through 5 dictate the velocities of manipulator joints 0 through 4 respectively. Column 6 specifies the length (seconds) of the move.

A DELAY statement is indicated by a "2." It suspends manipulator motion for a specified time segment, which is stored in column 6 of the instruction.

A WAITI statement is indicated by a "3." It commands a controller to wait indefinitely for a binary port to take on a particular value. During the execution of a WAITI instruction, manipulator motion is suspended. The parameters associated with this instruction are stored in columns 5 and 6. Column 5 specifies the port identification number while column 6 specifies the value to which it is compared.

A TESTI statement is indicated by a "4." It commands a controller to check the value of a port and to jump to another instruction, based on the value of the port. The parameters associated with this instruction are stored in columns 4 through 6. Column 4 specifies the port identification number. Column 5 specifies the value to be compared. Column 6 specifies the location of the instruction to jump to if the value of the port equals the value stored in column 5. If they are not equal, the subsequent instruction is executed.

A WRITEO statement is indicated by a "5." It instructs a controller to change the value of a binary port. Column 5 specifies the port identification number while column 6 specifies the value.

A JUMP statement is indicated by a "6." It commands a controller to unconditionally skip to the instruction, whose location is stored in column 6.

A SETC statement is indicated by a "7." It instructs a controller to change the value of a designated integer variable to the one stored in column 6.

A TESTC statement is indicated by an "8." It commands a controller to check the value of a designated integer variable and to jump to another instruction depending upon its value. Column 5 specifies the value to which the variable is compared. Column 6 stores the location of the instruction to jump to. If they are not equal, the subsequent instruction is executed.

The control programs used by robots 1 and 2 and machine tools 1 and 2 are presented in the following tables.

Robot 1 Control Program

Instruction Number	Instruction					
0	1	2	1	1.5	1.5	1
1	3	0	0	0	25	1
2	1	0	1	0	0	.9
3	1	0	0	0	-.5	1
4	1	0	0	1	0	.5
5	1	0	-1	0	0	.9
6	7	0	0	0	0	1
7	6	0	0	0	0	13
8	1	0	1	0	0	1.9
9	1	0	0	0	-.5	1
10	1	0	0	1	0	.5
11	1	0	-1	0	0	1.9
12	7	0	0	0	0	2
13	1	1	0	0	0	6.75
14	1	0	1	0	0	.9
15	1	0	0	-1	0	.5
16	1	0	0	0	.5	1
17	1	0	-1	0	0	.9
18	8	0	0	0	2	21
19	5	0	0	0	1	0
20	6	0	0	0	0	22
21	5	0	0	0	1	1
22	5	0	0	0	0	1
23	3	0	0	0	5	1
24	5	0	0	0	0	0
25	3	0	0	0	5	0
26	1	0	1	0	0	.9
27	1	0	0	0	-.5	1
28	1	0	0	1	0	.5
29	1	0	-1	0	0	.9
30	8	0	0	0	2	33
31	3	0	0	0	31	1
32	6	0	0	0	0	34
33	3	0	0	0	30	0
34	1	1	0	0	0	9.75
35	8	0	0	0	2	41
36	1	0	1	0	0	1.9
37	1	0	0	-1	0	.5
38	1	0	0	0	.5	1
39	1	0	-1	0	0	1.9
40	6	0	0	0	0	45
41	1	0	1	0	0	.9
42	1	0	0	-1	0	.5
43	1	0	0	0	.5	1
44	1	0	-1	0	0	.9
45	1	-1	0	0	0	16.5
46	8	0	0	0	1	8
47	6	0	0	0	0	1

Robot 2 Control Program

Instruction Number	Instruction					
0	1	2	1	1.5	1.5	1
1	1	1	0	0	0	13
2	3	0	0	0	30	1
3	1	-1	0	0	0	16.5
4	1	0	1	0	0	.9
5	1	0	0	0	-.5	1
6	1	0	0	1	0	.5
7	1	0	-1	0	0	.9
8	7	0	0	0	0	1
9	6	0	0	0	0	15
10	1	0	1	0	0	1.9
11	1	0	0	0	-.5	1
12	1	0	0	1	0	.5
13	1	0	-1	0	0	1.9
14	7	0	0	0	0	2
15	1	1	0	0	0	6.75
16	1	0	1	0	0	.9
17	1	0	0	-1	0	.5
18	1	0	0	0	.5	1
19	1	0	-1	0	0	.9
20	8	0	0	0	2	23
21	5	0	0	0	1	1
22	6	0	0	0	0	24
23	5	0	0	0	1	0
24	5	0	0	0	0	1
25	3	0	0	0	10	1
26	5	0	0	0	0	0
27	3	0	0	0	10	0
28	1	0	1	0	0	.9
29	1	0	0	0	-.5	1
30	1	0	0	1	0	.5
31	1	0	-1	0	0	.9
32	3	0	0	0	35	0
33	1	1	0	0	0	9.75
34	8	0	0	0	2	41
35	1	0	1	0	0	1.9
36	1	0	0	-1	0	.5
37	1	0	0	0	.5	1
38	1	0	-1	0	0	1.9
39	1	-1	0	0	0	16.5
40	6	0	0	0	0	10
41	1	0	1	0	0	.9
42	1	0	0	-1	0	.5
43	1	0	0	0	.5	1
44	1	0	-1	0	0	.9
45	6	0	0	0	0	2

Machine Tool 1 Control Program

Instruction Number	Instruction					
0	3	0	0	0	0	1
1	5	0	0	0	0	1
2	4	0	0	1	1	12
3	1	.5	.5	.5	0	1.8
4	1	0	.5	0	0	2.0
5	1	0	0	0	0	2.2
6	1	.16	0	0	0	8.125
7	1	0	0	0	0	2.2
8	1	-.5	-.5	0	0	3.8
9	1	-.5	0	0	0	.6
10	5	0	0	0	0	0
11	6	0	0	0	0	0
12	1	.5	.5	0	0	1.8
13	1	0	.5	0	0	2.3
14	1	0	0	1	0	2.2
15	1	.16	0	0	0	8.125
16	1	0	0	-1	0	2.2
17	1	-.5	-.5	0	0	4.1
18	1	-.5	0	0	0	.3
19	6	0	0	0	0	10

Machine Tool 1 Control Program

Instruction Number	Instruction					
0	3	0	0	0	15	1
1	5	0	0	0	0	1
2	4	0	0	16	1	12
3	1	.5	.5	0	0	1.8
4	1	0	.5	0	0	.3
5	1	0	0	1	0	2.1
6	1	.16	0	0	0	8.125
7	1	0	0	-1	0	2.1
8	1	-.5	-.5	0	0	2.1
9	1	-.5	0	0	0	2.3
10	5	0	0	0	0	0
11	6	0	0	0	0	0
12	1	.5	.5	0	0	1.8
13	1	0	.5	0	0	.3
14	1	0	0	1	0	2.1
15	1	.16	0	0	0	8.125
16	1	0	0	-1	0	2.1
17	1	-.5	-.5	0	0	2.1
18	1	-.5	0	0	0	2.3
19	5	0	0	0	0	0
20	6	0	0	0	0	0

APPENDIX B SUB-MODEL CLASSES

B.0 Introduction

This appendix describes the master simulation, internal interaction, external interaction, and passive sub-model classes utilized in the test bed models. These classes execute in a single process configuration and were created using the C programming language. Each are discussed separately.

B.1 Master Simulation Sub-Model Class

The master simulation class is implemented as a main subroutine in an executable file. It integrates the logic flow of the model demonstration environment described in 5.2 with the logic flow of the master simulation class described in 4.2.0.1.1 and 4.2.2.

The class executes the model selection phase by illustrating twenty-five icons, which represent the twenty-five test bed models, on the CRT screen and by requesting a user to select a model. Upon model selection, the class locates the corresponding initialization file and enters the simulation initialization phase.

Once the various sub-models are initialized along with CLOCK, SYNCH_PORT, I/O_PORT, and DATA_TABLE, the class enters the model execution phase, where it iteratively executes the clock update, sub-model scan, and model control phases. During the model control phase, the class interacts with a user in the fashion described in 5.2. When the conditions for simulation termination have been met, it terminates both the simulation and the demonstration environment. An output phase is not executed.

B.2 Internal Interaction Sub-Model Classes

There are two internal interaction classes used in the test bed models. They are Discrete Manipulator and Kin/Geo Manipulator. Both classes execute the internal interaction logic flow described in 4.2.0.1.2, and each is described separately.

B.2.0 Discrete Manipulator

The Discrete Manipulator class utilizes discrete event simulation to represent an intelligent resource. By intelligent, it is meant that the resource executes an alterable program. As a result, the class executes an alterable program, encoded in a data structure, to dictate how it responds to its environment.

The main attributes of this class are its instruction set, instruction pointer, allocated ports, busy/idle status, and future occurrence time. The instruction set utilizes various types of statements. These include DELAY, WAITI, TESTI, WRITEO, JUMP, SETC, and TESTC. The methods in which these statements are interpreted are similar to those described in A.3.

The instruction pointer contains the address of the instruction (activity) that is currently being executed.

The allocated ports are the group of I/O_PORT records that the class is permitted to change.

The busy/idle status indicates whether or not the class is executing a WAITI instruction. If it is, it is considered idle, otherwise it is considered busy. The future occurrence time indicates the completion time of the current instruction.

After the class executes the clock scan phase, it determines whether the current instruction is a DELAY statement. If it is, it then checks whether the new value of CLOCK equals the completion time of this statement. If they are equal, the class executes as many conditional instructions as possible before encountering another DELAY statement or an unsatisfied WAITI statement. If it is not, the class does not activate and reports its completion in SYNCH_PORT.

If the class is not currently executing a DELAY statement, it executes as many conditional instructions as possible before encountering a DELAY or WAITI statement. If a DELAY statement is encountered, then its future occurrence time is set to the completion time of the statement. If an unsatisfied WAITI statement is encountered, its future occurrence time is set to infinity.

Note that the execution of a DELAY statement is a time-based activity, while the execution of the other instructions utilizes conditional based activities. In general, the execution of a DELAY statement simulates a resource performing an activity, while the execution of the other statements dictates which activities are simulated.

B.2.1 Kin/Geo Manipulator

The Kin/Geo Manipulator class utilizes continuous simulation and geometric modeling to represent intelligent manipulators. The main attributes of this sub-model class are its instruction set, instruction pointer, kinematic description vectors, geometric models, and its future occurrence time. The instruction set utilizes various types of statements. These include PMOVE, DELAY, WAITI, TESTI, WRITEO, JUMP, SETC, and TESTC. The methods in which these instructions are interpreted are similar to those described in A.3. Note that the execution of a PMOVE statement simulates joint and link motion.

The instruction pointer holds the address of the instruction currently being executed.

The kinematic description vectors include joint position vectors, joint velocity vectors, link position vectors, and link velocity vectors. These vectors and their relationships are described in A.2. The geometric models are solids models of the manipulator links.

The future occurrence time is always equal to infinity or the value of CLOCK plus a fixed time increment.

After the class executes the clock scan phase, it checks whether it is currently executing either a DELAY statement or a PMOVE statement. If it is executing a DELAY statement, it checks whether its completion time is less than or equal to the new value of CLOCK. If it is, it simulates the passage time, and executes as many conditional instructions as possible before encountering a DELAY statement, PMOVE statement, or unsatisfied WAITI statement. If it is not, it simulates the passage of time and indicates its completion in SYNCH_PORT. The execution of a PMOVE statement works in a similar fashion with the exception that it simulates manipulator motion.

If neither a DELAY statement nor PMOVE statement is currently being executed, the class attempts to execute as many conditional instructions as possible before encountering a PMOVE statement, DELAY statement, or unsatisfied WAITI statement.

The execution of a DELAY statement is a time-based activity while the execution of a PMOVE statement utilizes attribute description functions. The other instructions are conditional activities.

B.3 External Interaction Sub-Model Classes

There are eighteen external interaction sub-model classes that are utilized by the test bed models. These classes are group into five categories:

1. arrival and departure,
2. grasp and release,
3. transport,
4. cut, and
5. sense.

All classes execute the logic flow described in 4.2.0.1.3 and each are discussed separately within their categories..

B.3.0 Arrival and Departure

The classes in the arrival and departure category are responsible for the arrival of part (passive) sub-models into a simulation along with their removal. There are four classes:

1. Discrete Part Arrival,
2. Kin/Geo Part Arrival,
3. Discrete Part Removal, and
4. Kin/Geo Part Removal.

All four classes utilize time-based activities, namely the time of part arrival or departure. In addition, all four classes are independent of other classes used in the test bed models.

The Discrete Part Arrival class monitors CLOCK and checks whether it is equal to the arrival time of the next batch of type "0" parts. If it is, the class creates two passive sub-model data structures, attaches symbolic type and location attributes, and places the addresses of the sub-models into the interaction buffer of pallet 1. It then computes the arrival time of the next batch of type "0" parts. Note that the inter-arrival time for these parts is 60 seconds. If CLOCK is not equal to the arrival time, the class indicates its completion in SYNCH_PORT.

The Kin/Geo Part Arrival class performs in an identical fashion with the exception that it adds kinematic and geometric attributes to the passive sub-models rather than symbolic ones.

The Discrete Part Removal class monitors CLOCK and checks whether it is equal to the departure time of parts resident on pallet 3. If its is, the class finds the addresses of the part sub-models in the interaction buffer of pallet 3 and removes them from a simulation by disallocating their memory. It then computes the departure time of the next batch of part sub-models.

The Kin/Geo Part Removal class works in a similar fashion, with the exception that it disallocates more memory (due to the kinematic and geometric attributes of the part sub-models).

B.3.1 Grasp and Release

Classes in the grasp and release category are responsible for transferring the possession of part sub-models between robot sub-models and machine tool and pallet sub-models. There are eight classes:

1. Discrete Part Grasp,
2. Kin/Geo Part Grasp,
3. Discrete to Kin/Geo Part Grasp,
4. Kin/Geo to Discrete Part Grasp,
5. Discrete Part Release,
6. Kin/Geo Part Release,
7. Discrete Part to Kin/Geo Part Release, and
8. Kin/Geo to Discrete Part Release.

All classes are dependent upon the execution of robot sub-models which utilize either the Discrete Manipulator or Kin/Geo Manipulator classes. Once the appropriate robot sub-models have executed, the grasp and release classes check their instruction pointers to determine if they are performing grasp or release operations. If they are, the grasp classes transfer possession of part sub-models from machine tool sub-models or pallet sub-models to their respective robot sub-models. This involves the transfer of part sub-model addresses between respective interaction buffers. Likewise, release classes transfer possession of part sub-models from their respective robot sub-models to either machine tool sub-models or a pallet sub-models.

Discrete Part Grasp and Discrete Part Release classes handle part transfers for robot sub-models which utilize Discrete Manipulator classes and for source and destination sub-models which require part sub-models that utilize symbolic attributes. Kin/Geo Grasp and Kin/Geo Release classes handle part transfers for robot sub-models which utilize Kin/Geo Manipulator classes and for source and destination sub-model which require part sub-models that utilize kinematic and geometric attributes.

Discrete to Kin/Geo Part Grasp classes are used in conjunction with robot sub-models which utilize the Kin/Geo Manipulator class. They expand the attribute sets of part sub-models from symbolic parameters to kinematic and geometric parameters as well as transfer their possession. The class expands an attribute set by finding the kinematic and geometric attributes in DATA_TABLE which correspond to the symbolic attributes of the part sub-model, and by allocating memory for the new attributes.

The Discrete to Kin/Geo Part Release class works in a similar fashion with the exception that it transfers possession of a part sub-model from a robot sub-model, which utilizes a Discrete Manipulator class, to a sub-model which requires that interacting part sub-models utilize kinematic and geometric attributes.

The Kin/Geo to Discrete Part Grasp and Kin/Geo to Discrete Part Release classes are mirror images of the Discrete to Kin/Geo Part Grasp and Discrete to Kin/Geo Part Release classes respectively. During part sub-model transfers, they contract the attribute sets of part sub-models from kinematic and geometric parameters to symbolic parameters. To do so, they find the symbolic parameters in DATA_TABLE which correspond to the kinematic and geometric parameters of the part sub-models, assign these values to the existing symbolic attributes of sub-models, and then disallocate memory associated with the kinematic and geometric attributes.

Note that all of the classes just described execute conditional activities only. As a result, their future occurrence times are always set to infinity.

B.3.2 Transport

The Kin/Geo Part Transport class is responsible for updating the kinematic attributes of part sub-models that are possessed by robot and machine tool sub-models that utilize the Kin/Geo Manipulator class. This class is dependent upon the execution of robot sub-models.

Once an appropriate sub-model has executed, the class checks its instruction pointer to determine if it is currently executing a transport operation. If it is, the class determines the change in simulated time and checks the kinematic attributes of the robot gripper or machine tool bed. It then computes the new kinematic values for the part sub-model currently possessed by the sub-model. In general, this class executes both conditional activities and attribute description functions. However, its future occurrence time is always set to infinity.

B.3.3 Cut

Classes belonging to the cut category are responsible for simulating the cutting of a part by a machine tool cutter. There are two classes: Discrete Part Cut and Kin/Geo Part Cut.

The Discrete Part Cut class is dependent upon the execution of machine tool sub-models which utilize the Discrete Manipulator class. After the execution of a machine tool sub-model, the class checks its instruction pointer to determine if it has just completed a cutting operation. If it has, the class updates the symbolic type of the part sub-model currently possessed by the machine tool sub-model. This process involves the execution of conditional activities.

The Kin/Geo Part Cut class is dependent upon the execution of machine tool sub-models which utilize the Kin/Geo Manipulator class along with their associated transport sub-models. When a machine tool sub-model executes a sequence of cutting operations and a transport sub-model updates the kinematic attributes of the part sub-model that it possesses, the class subtracts the solids model representing the machine tool cutter from the solids model representing the part. This process involves the execution of both conditional activities and attribute description functions. Note that the future occurrence times for both classes are always set to infinity.

B.3.4 Sense

The Discrete Part Sensor class is responsible for detecting the presence of part sub-models at pallet sub-models and transmitting messages to I/O_PORT. Depending upon its application, it is dependent upon sub-models utilizing classes in the arrival and departure category and/or grasp and release category.

After an appropriate sub-model has executed, the class checks for the presence of parts by examining the contents of the pallet interaction buffer. If the pallet interaction buffer is full, the class provides a "high" value to one of its two allocated I/O_PORT records. If the buffer is empty, the class provides a "high" value to its other port record. If the buffer is neither full nor empty, the class provides both ports with "low" values. This process involves the execution of conditional activities. The future occurrence time of the class is always set to infinity.

B.4 Passive Sub-Model Classes

There are two passive sub-model classes used in the test bed models. They are part and pallet. The part class is dynamic and contains two sets of attributes. The first set contains a symbolic location and part type. Both of these attributes are integers. The second set contains the kinematic parameters described in A.2 and a solids model.

The pallet class is static and contains an interaction buffer capable of holding the addresses of two part sub-models. The attribute set of the class includes the current number of part sub-models that its possesses, the kinematic parameters described in A.2, and a solids model.

APPENDIX C TEST BED SUB-MODELS

C.0 Introduction

Sixty-four active sub-models are utilized in the test bed models. This appendix identifies these sub-models along with their classes.

C.1 Master Simulation Sub-Model Class

1. **MS**: executes the master simulation sub-model logic
   for the test bed models

C.2 Discrete Manipulator Class

1. **W1D**: simulates work cell 1
2. **W2D**: simulates work cell 2
3. **R1D**: simulates robot 1
4. **R2D**: simulates robot 2
5. **M1D**: simulates machine tool 1
6. **M2D**: simulates machine tool 2

C.3 Kin/Geo Manipulator Class

1. **R1K**: simulates robot 1
2. **R2K**: simulates robot 2
3. **M1K**: simulates machine tool 1
4. **M2K**: simulates machine tool 2

C.4 Discrete Part Arrival Class

1. **DPA**: simulates the arrival type "0" parts on pallet 1

C.5 Kin/Geo Part Arrival Class

1. **KGPA**: simulates the arrival type "0" parts on pallet 1

C.6 Discrete Part Removal Class

1. **DPD**: simulates the removal of parts from pallet 3

C.7 Kin/Geo Part Removal Class

1. **KGPD**: simulates the removal of parts from pallet 3

C.8 Discrete Part Grasp Class

1. **GDP1W1D**: simulates work cell 1 grasping a part from pallet 1
2. **GDP2W2D**: simulates work cell 2 grasping a part from pallet 2
3. **GDP1R1D**: simulates robot 1 grasping a part from pallet 1
4. **GDM1R1D**: simulates robot 1 grasping a part from machine tool 1
5. **GDP2R2D**: simulates robot 2 grasping a part from pallet 2
6. **GDM2R2D**: simulates robot 2 grasping a part from machine tool 2

C.9 Kin/Geo Part Grasp Class

1. **GKP1R1K**: simulates robot 1 grasping a part from pallet 1
2. **GKM1R1K**: simulates robot 1 grasping a part from machine tool 1
3. **GKP2R2K**: simulates robot 2 grasping a part from pallet 2
4. **GKM2R2K**: simulates robot 2 grasping a part from machine tool 2

C.10 Discrete Part Release Class

1. **RDW1P2D**: simulates work cell 1 releasing a part to pallet 2
2. **RDW2P3D**: simulates work cell 2 releasing a part to pallet 3
3. **RDR1P2D**: simulates robot 1 releasing a part to pallet 2
4. **RDR1M1D**: simulates robot 1 releasing a part to machine tool 1
5. **RDR2P3D**: simulates robot 2 releasing a part to pallet 3
6. **RDR2M2D**: simulates robot 2 releasing a part to machine tool 2

C.11 Kin/Geo Part Release Class

1. **RKR1P2K**: simulates robot 1 releasing a part to pallet 2
2. **RKR1M1K**: simulates robot 1 releasing a part to machine tool 1
3. **RKR2P3K**: simulates robot 2 releasing a part to pallet 3
4. **RKR2M2K**: simulates robot 2 releasing a part to machine tool 2

C.12 Discrete to Kin/Geo Part Grasp Class

1. **GDP1R1K**: simulates robot 1 grasping a part from pallet 1
2. **GDM1R1K**: simulates robot 1 grasping a part from machine tool 1
3. **GDP2R2K**: simulates robot 2 grasping a part from pallet 2
4. **GDM2R2K**: simulates robot 2 grasping a part from machine tool 2

C.13 Discrete to Kin/Geo Part Release Class

1. **RDR1P2K**: simulates robot 1 releasing a part to pallet 2
2. **RDR1M1K**: simulates robot 1 releasing a part to machine tool 1
3. **RDR2P3K**: simulates robot 2 releasing a part to pallet 3
4. **RDR2M2K**: simulates robot 2 releasing a part to machine tool 2

C.14 Kin/Geo to Discrete Part Grasp Class

1. **GKP1R1D**: simulates robot 1 grasping a part from pallet 1
2. **GKM1R1D**: simulates robot 1 grasping a part from machine tool 1
3. **GKP2R2D**: simulates robot 2 grasping a part from pallet 2
4. **GKM2R2D**: simulates robot 2 grasping a part from machine tool 2

C.15 Kin/Geo to Discrete Part Release Class

1. **RKR1P2D**: simulates robot 1 releasing a part to pallet 2
2. **RKR1M1D**: simulates robot 1 releasing a part to machine tool 1
3. **RKR2P3D**: simulates robot 2 releasing a part to pallet 3
4. **RKR2M2D**: simulates robot 2 releasing a part to machine tool 2

C.16 Kin/Geo Part Transport Class

1. **TKR1**: simulates robot 1 transporting a part
2. **TKR2**: simulates robot 2 transporting a part
3. **TKM1**: simulates machine tool 1 transporting a part
4. **TKM2**: simulates machine tool 2 transporting a part

C.17 Discrete Part Cut Class

1. **CDW1**: simulates work cell 1 cutting a part
2. **CDW2**: simulates work cell 2 cutting a part
3. **CDM1**: simulates machine tool 1 cutting a part
4. **CDM2**: simulates machine tool 2 cutting a part

C.18 Kin/Geo Part Cut Class

1. **CKM1**: simulates machine tool 1 cutting a part
2. **CKM2**: simulates machine tool 2 cutting a part

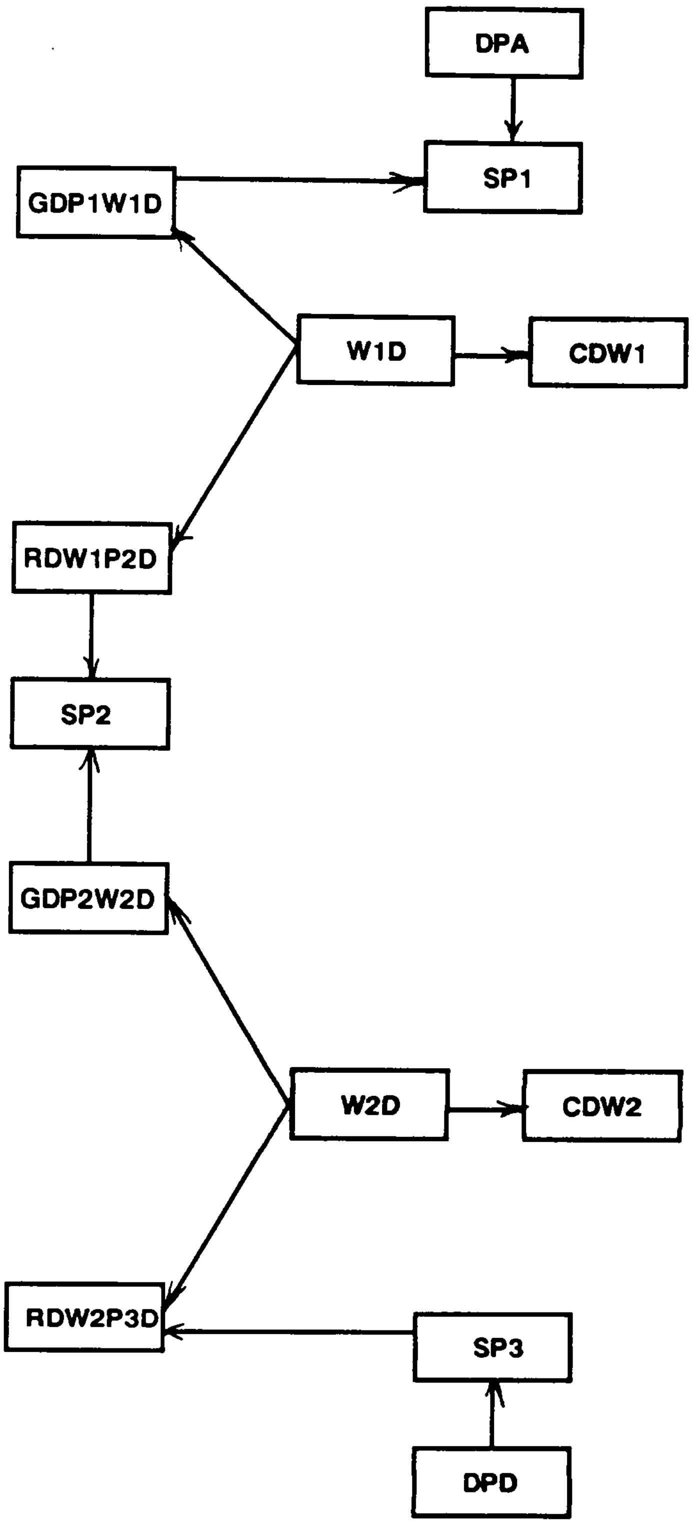
C.19 Discrete Part Sensor Class

1. **SP1**: simulates the limits switches at pallet 1
2. **SP2**: simulates the limits switches at pallet 2
3. **SP3**: simulates the limits switches at pallet 3

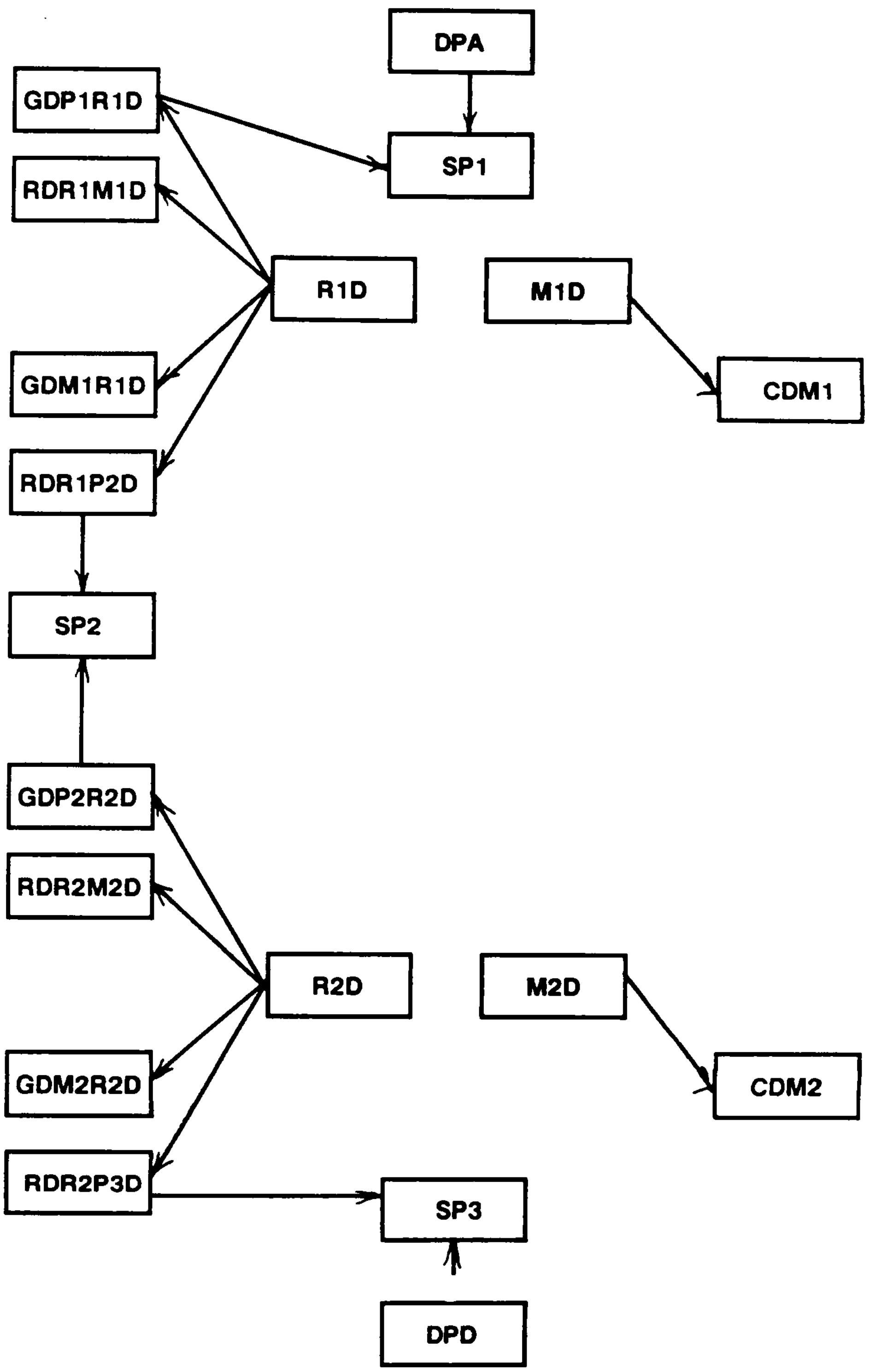
APPENDIX D TEST BED MODELS

D.0 Introduction

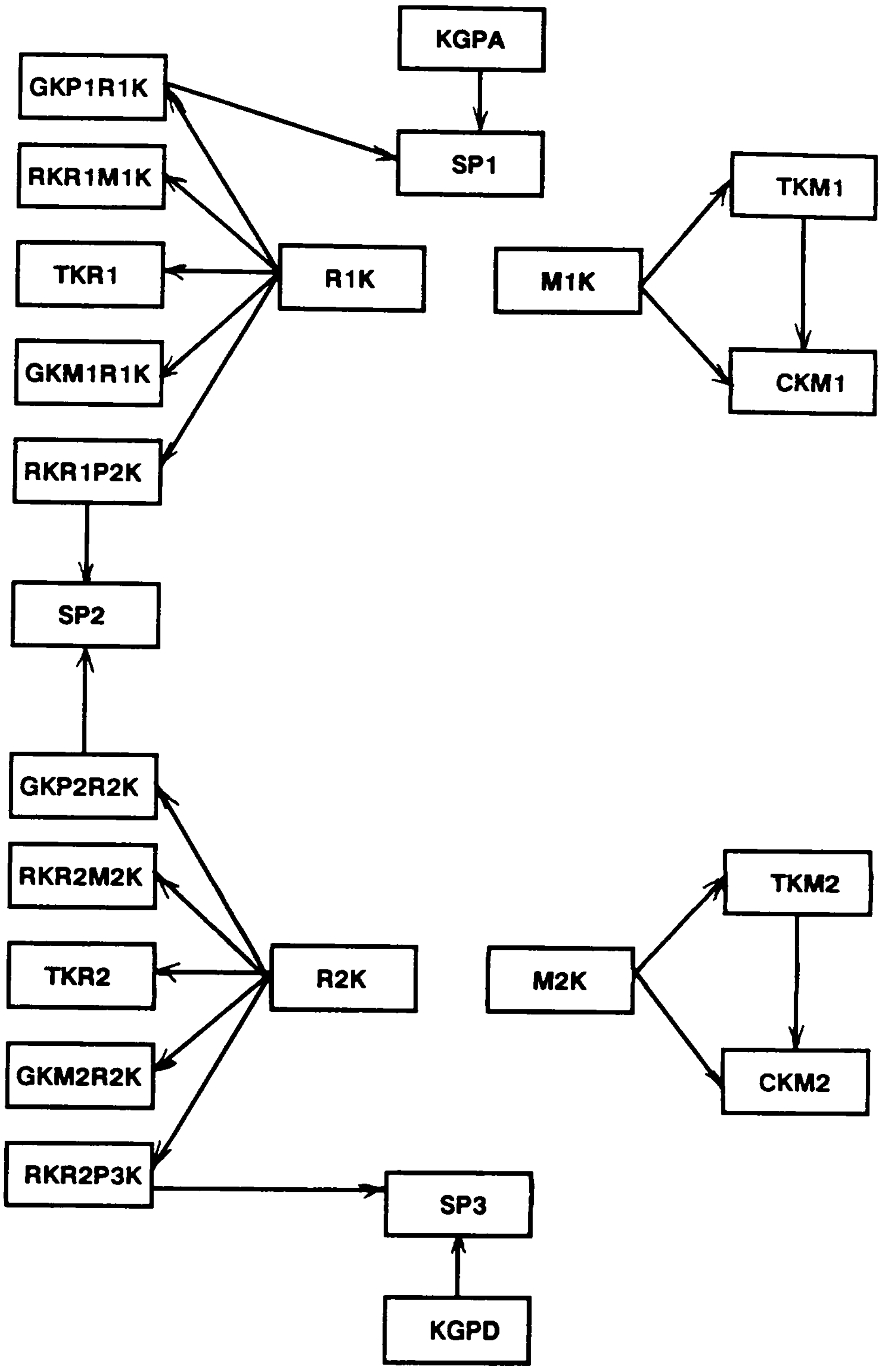
This appendix presents the sub-model structures of the twenty-five test bed models developed in this dissertation. Note that the figures do not show passive sub-models and that lines drawn between sub-models represent dependencies.



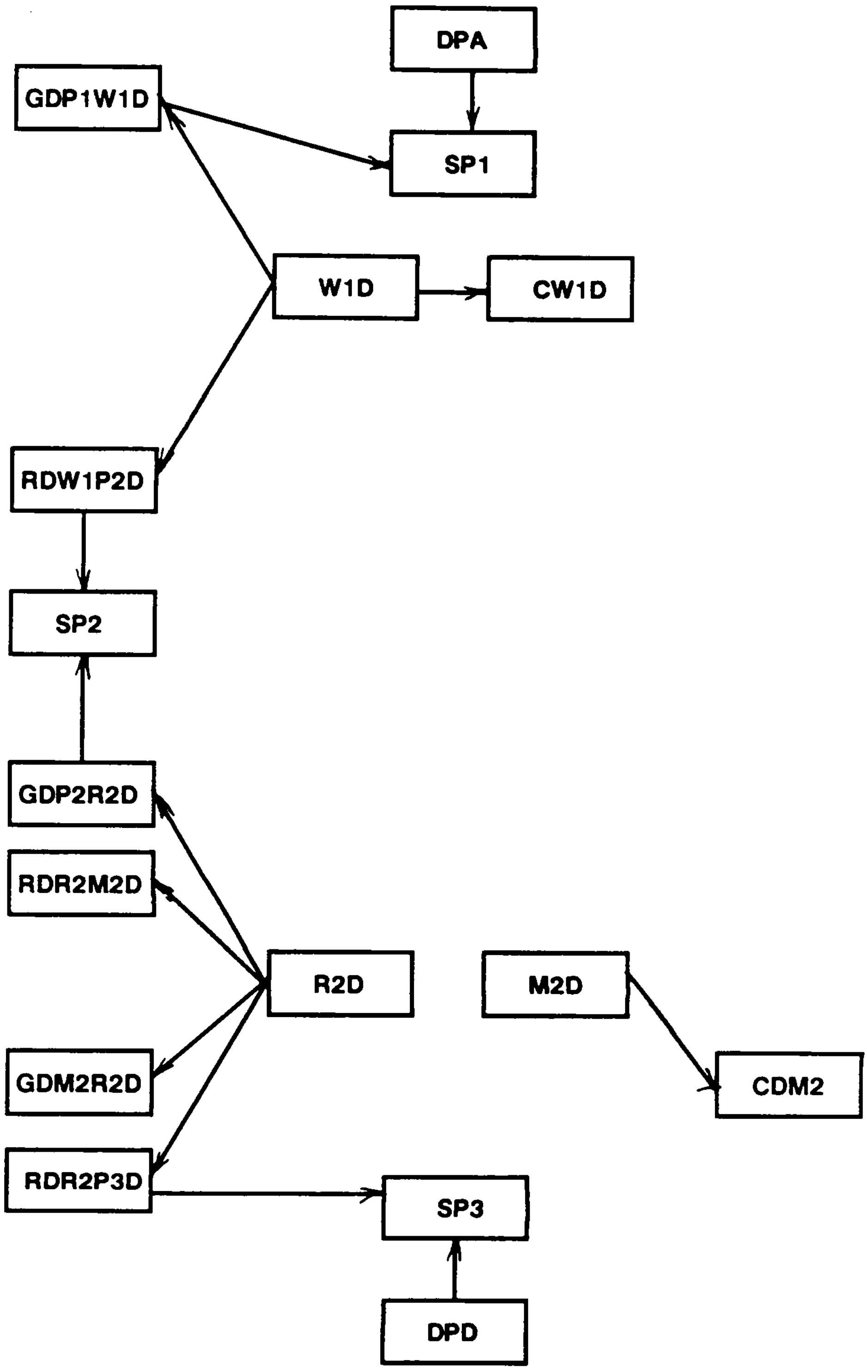
Test Bed Model 1



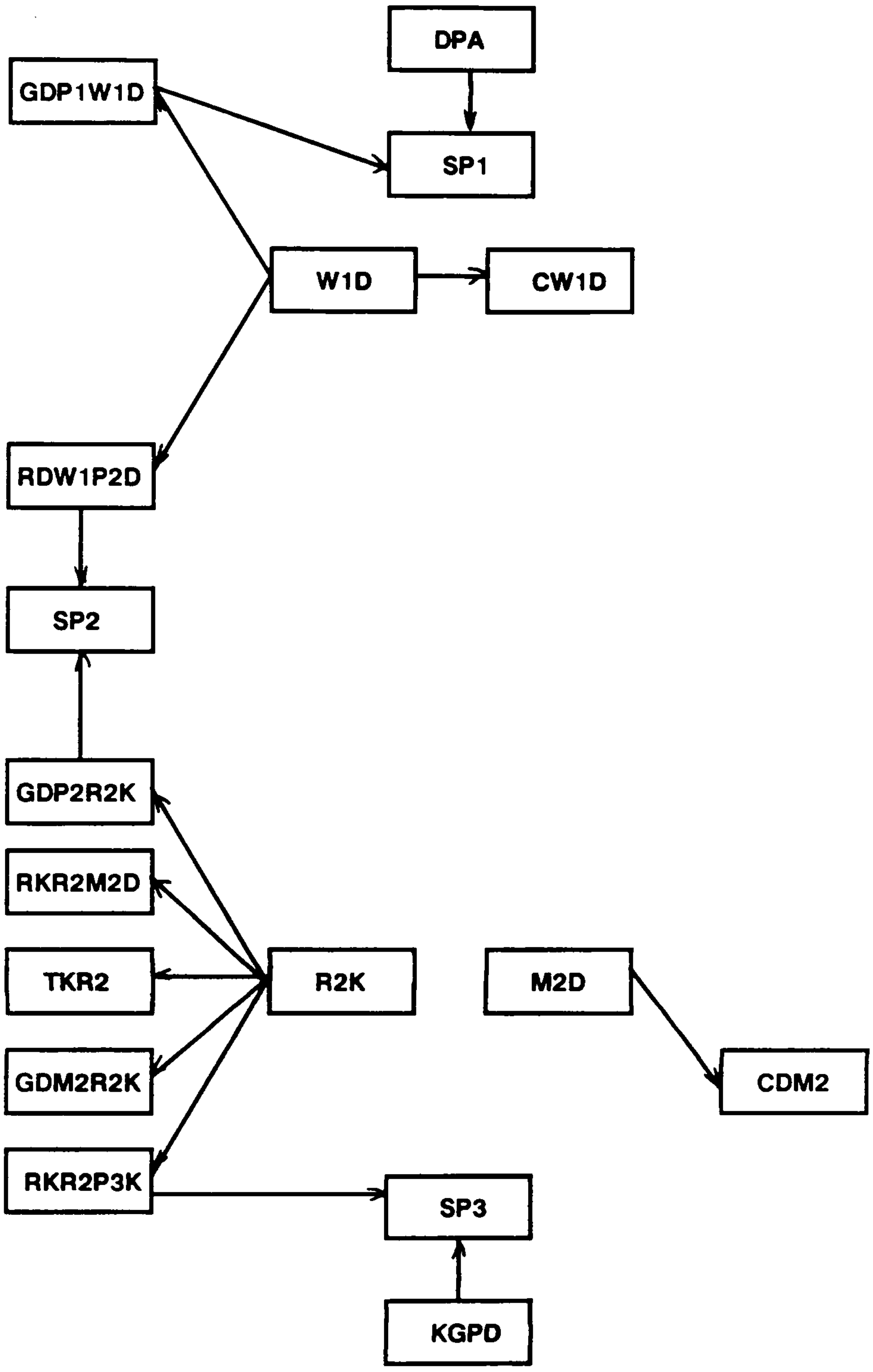
Test Bed Model 2



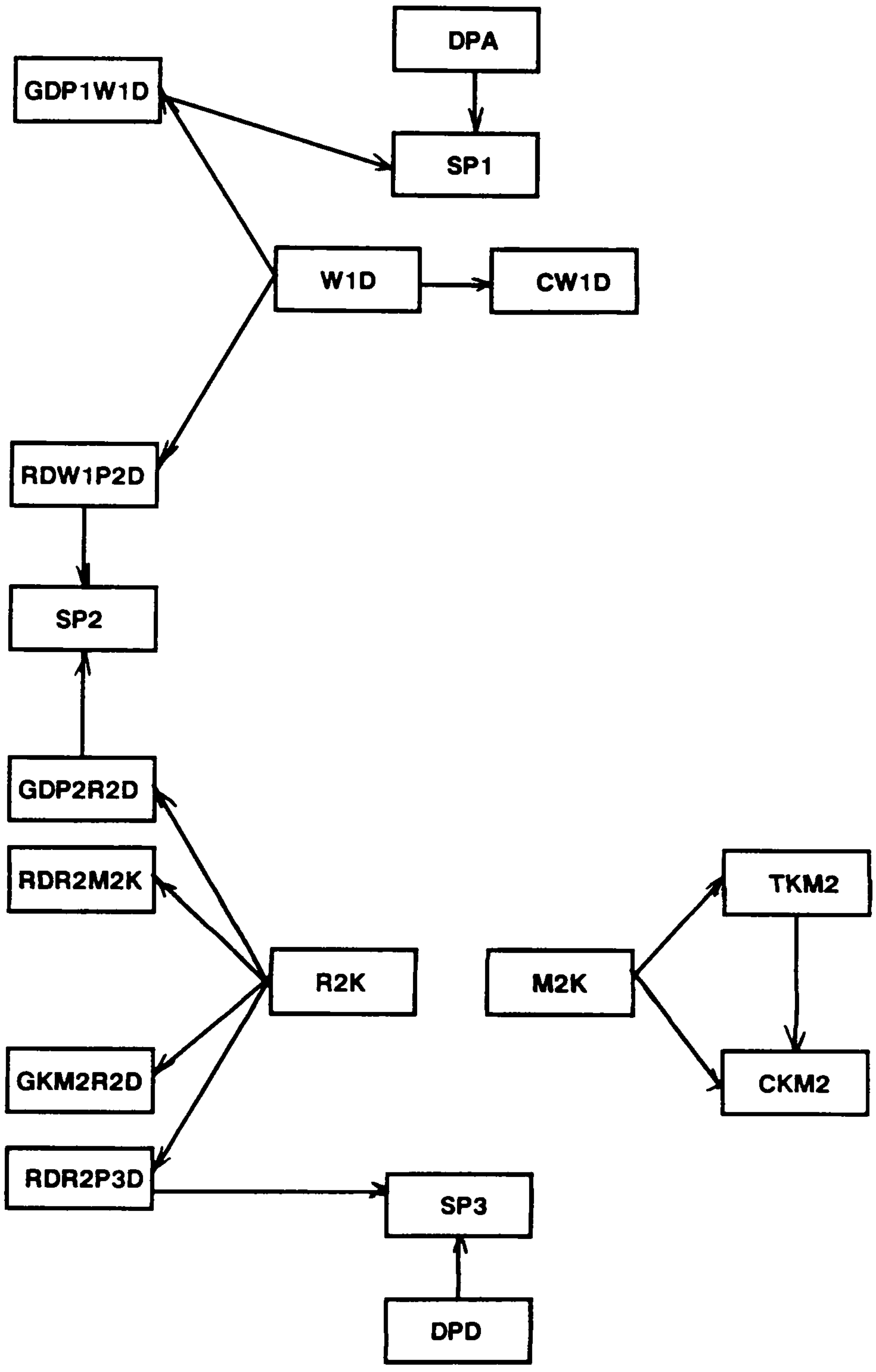
Test Bed Model 3



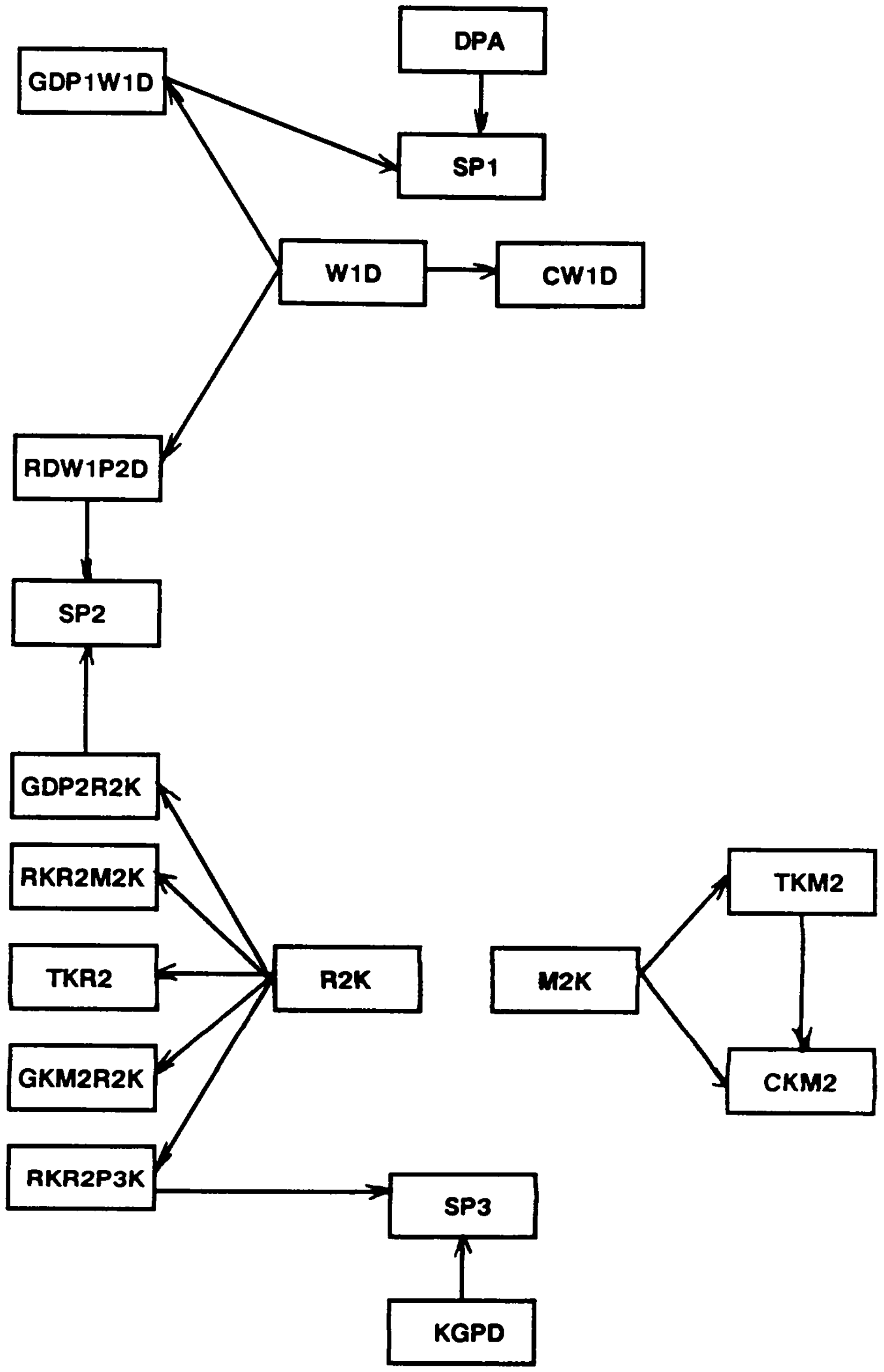
Test Bed Model 4



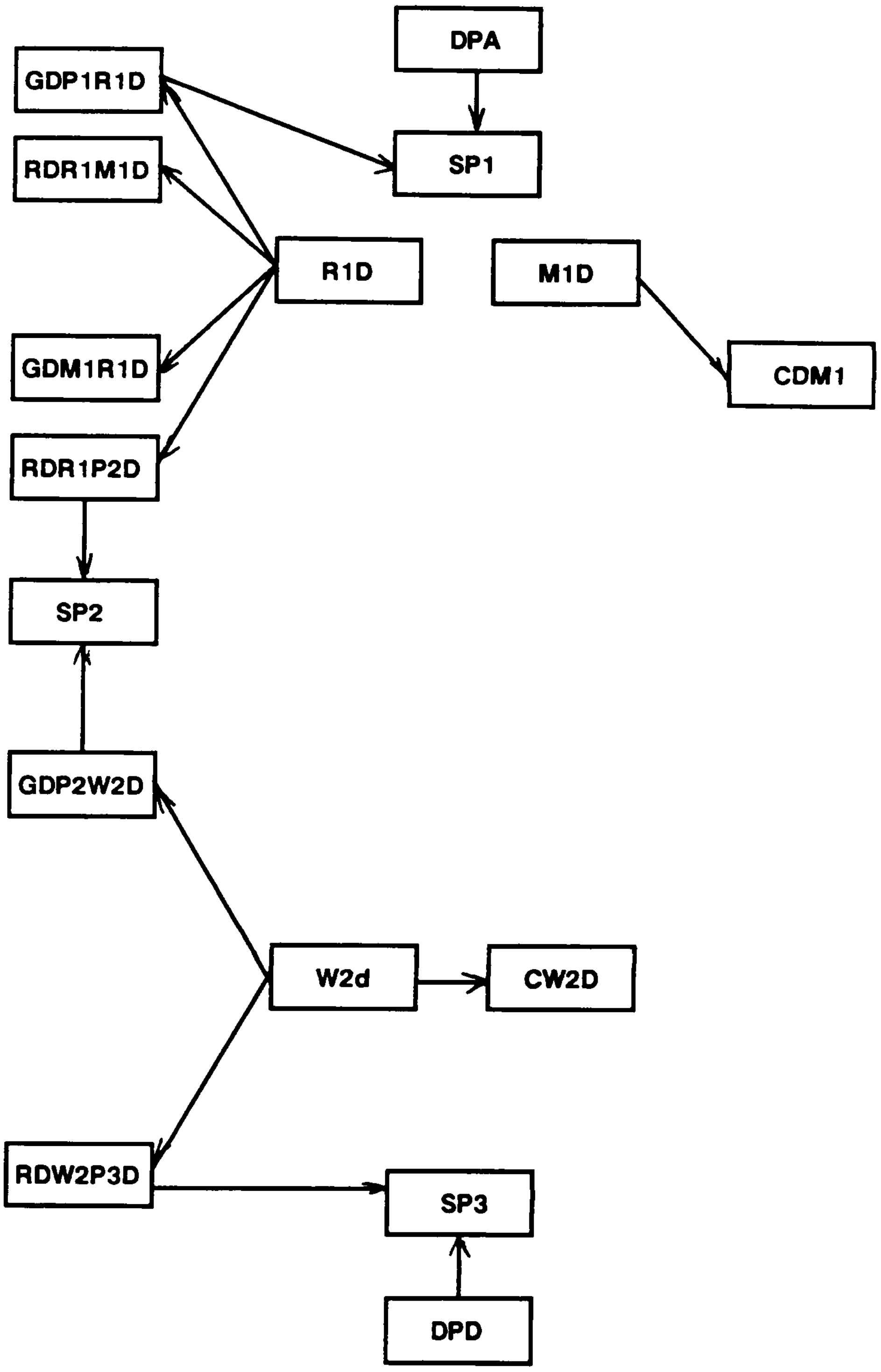
Test Bed Model 5



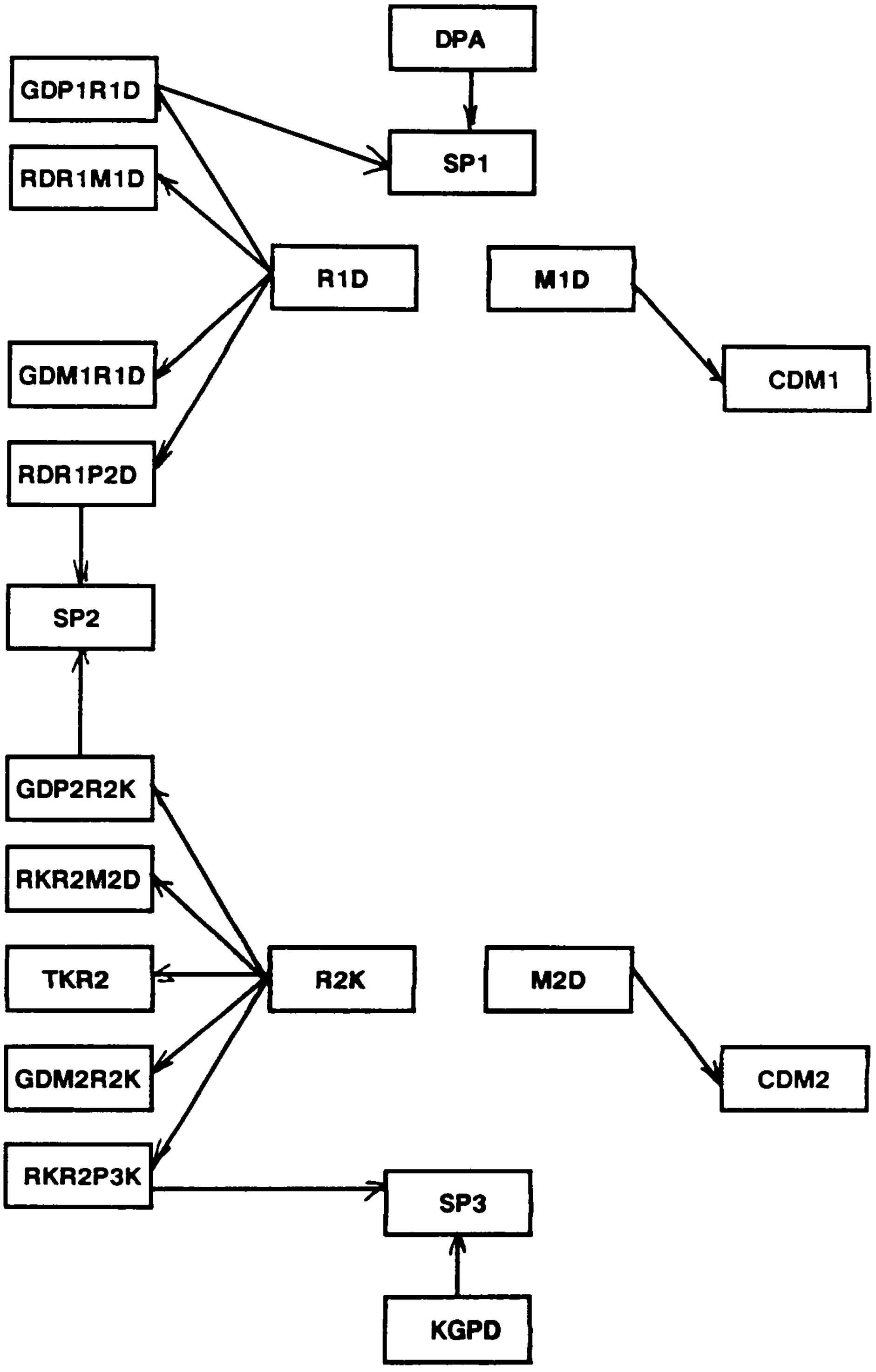
Test Bed Model 6



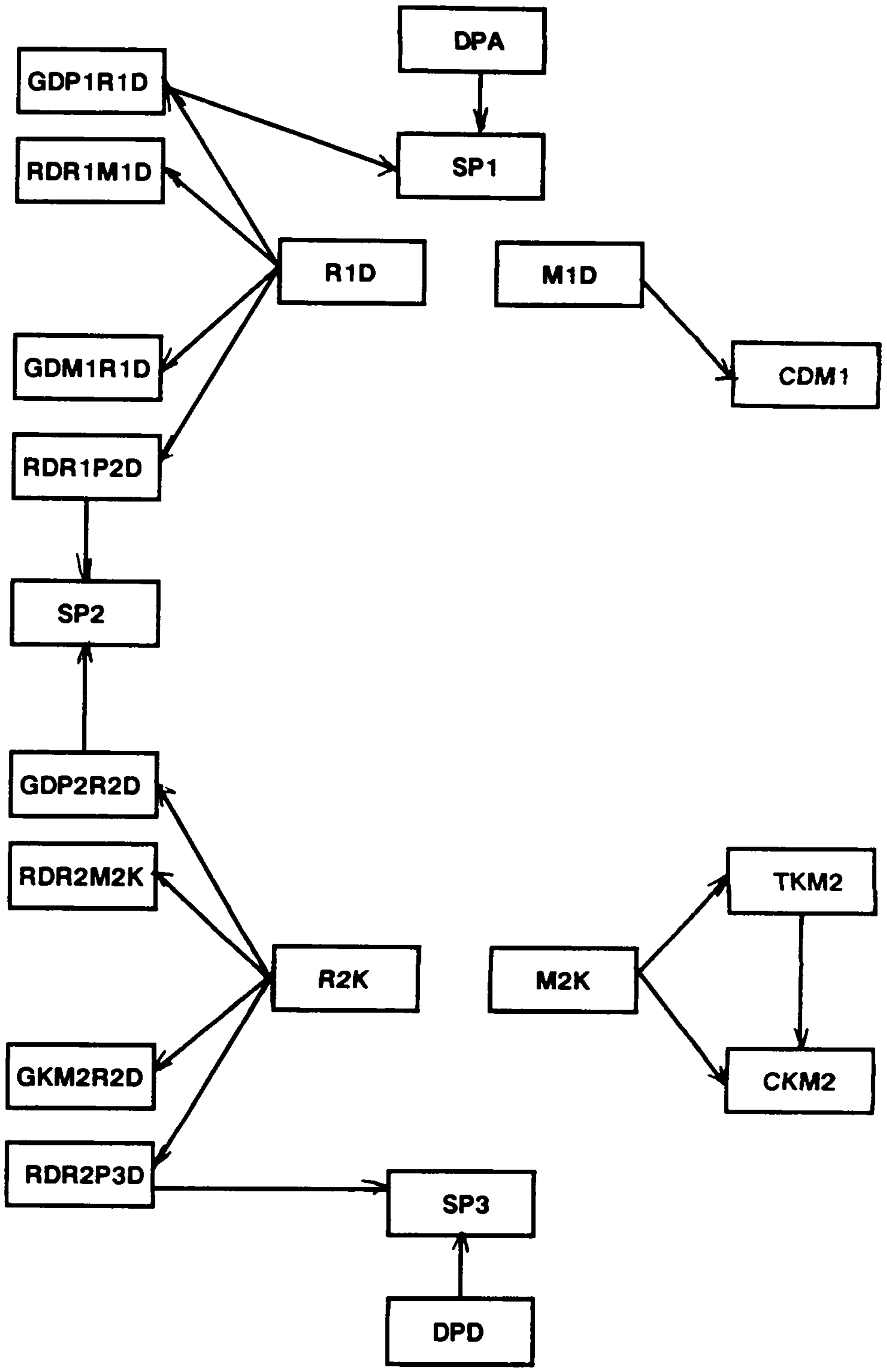
Test Bed Model 7



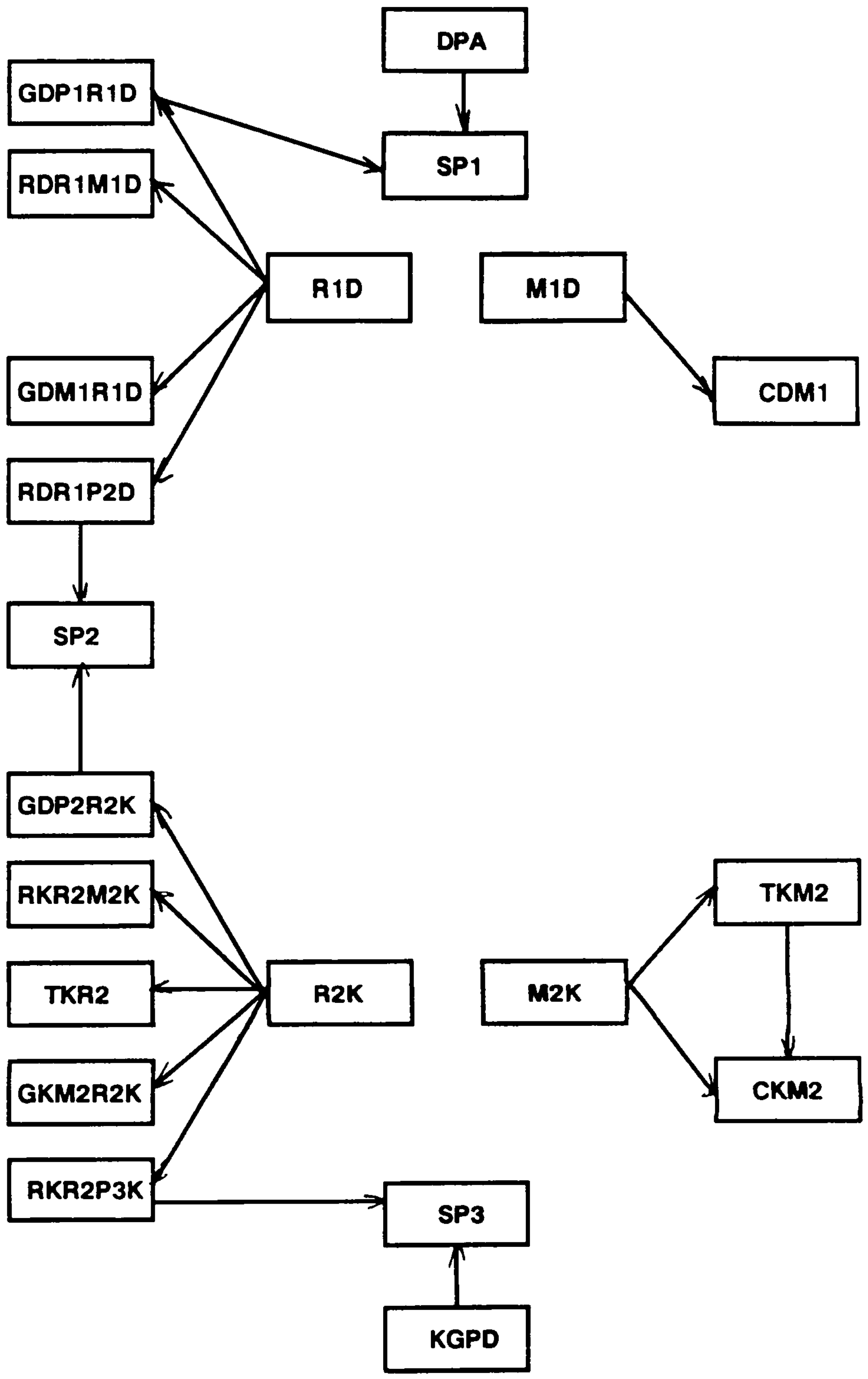
Test Bed Model 8



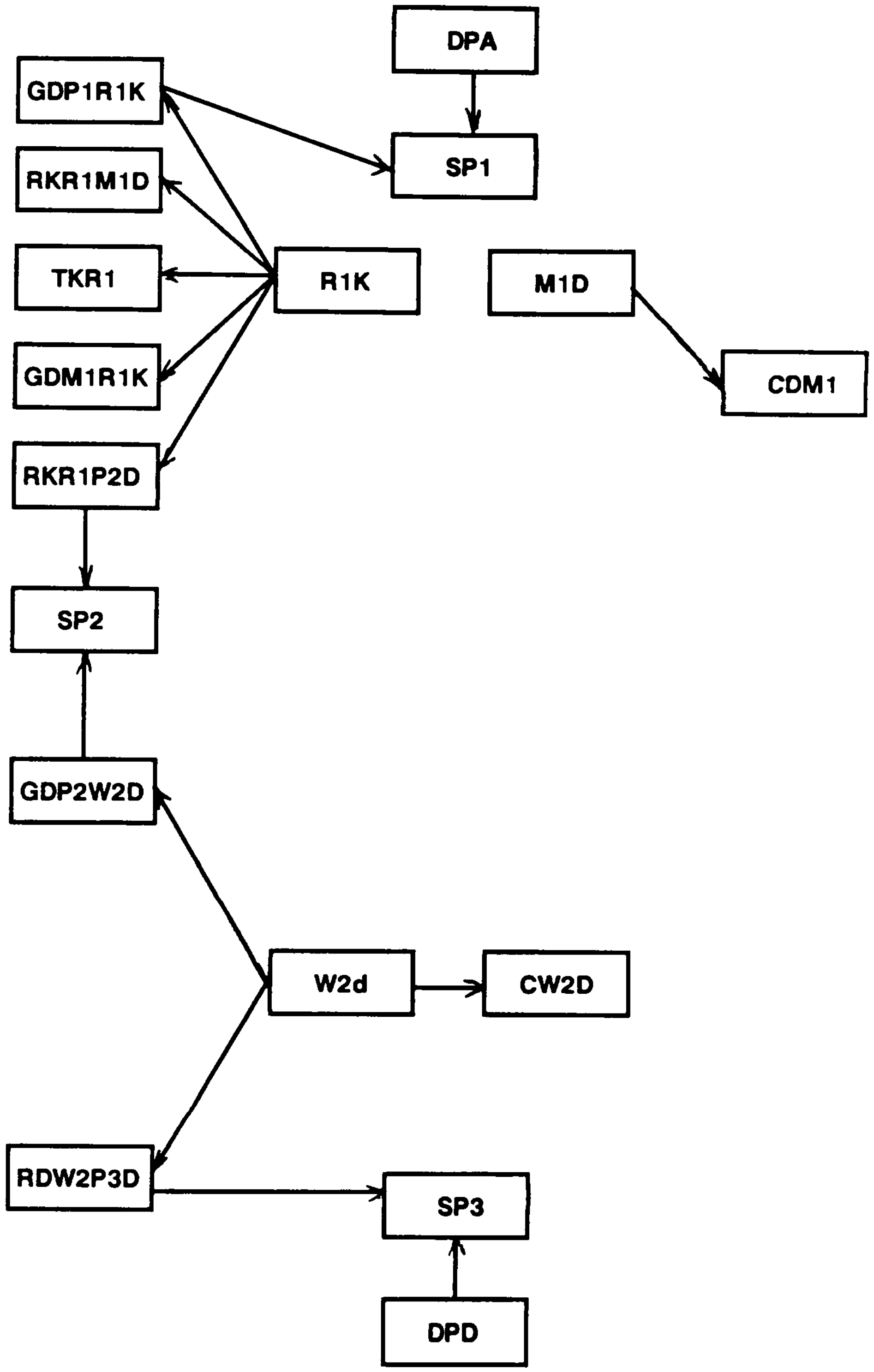
Test Bed Model 9



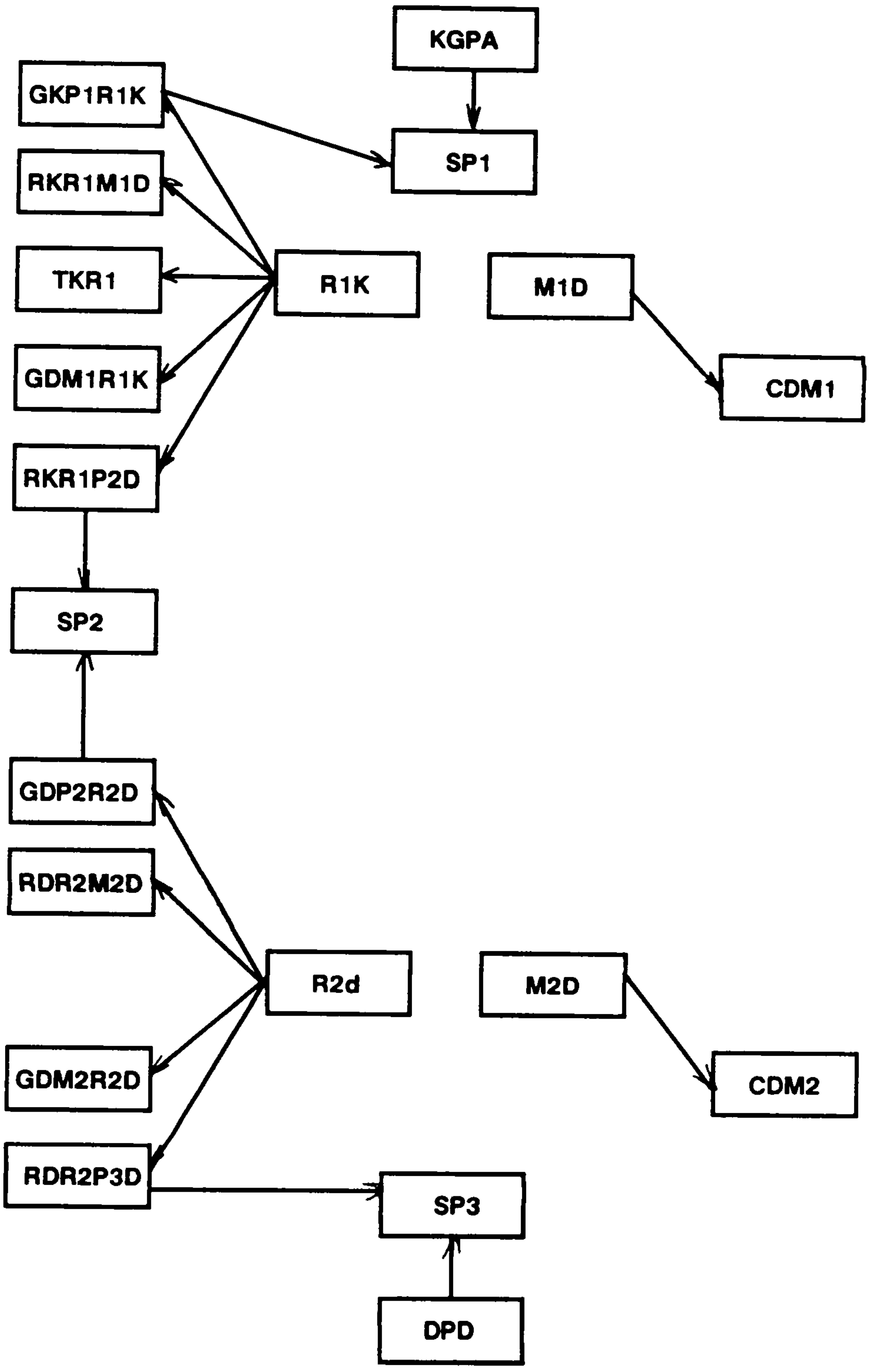
Test Bed Model 10



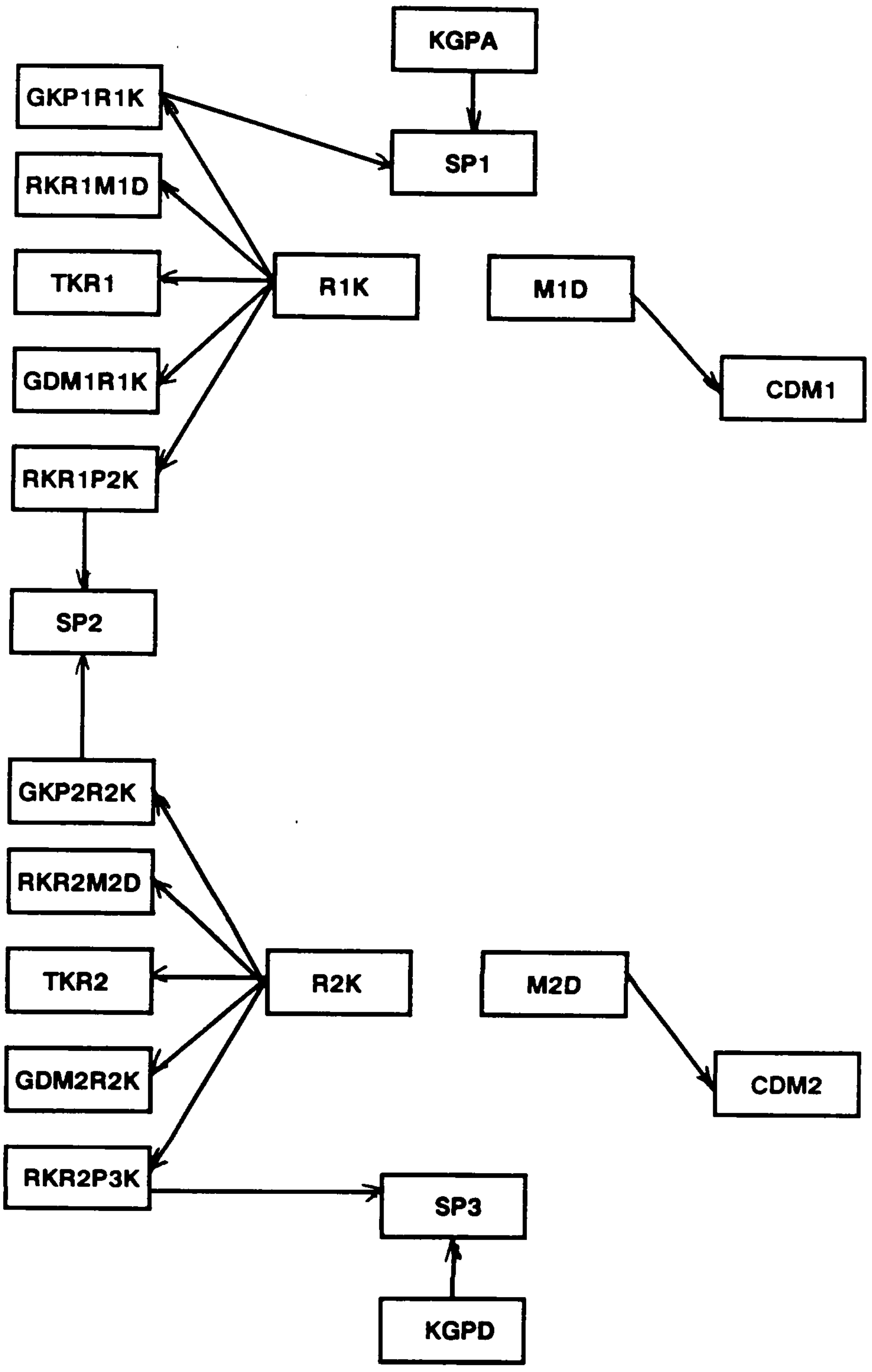
Test Bed Model 11



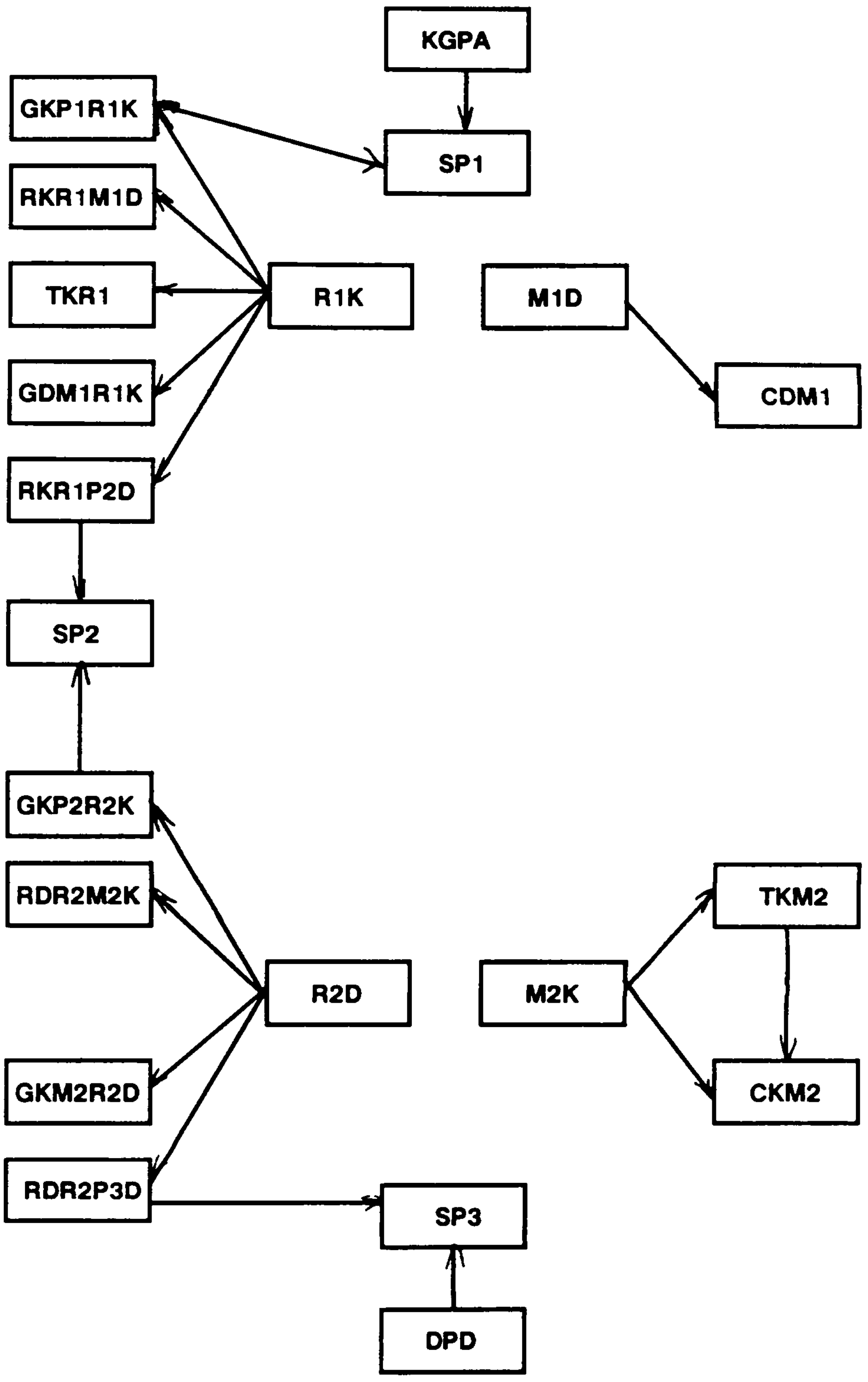
Test Bed Model 12



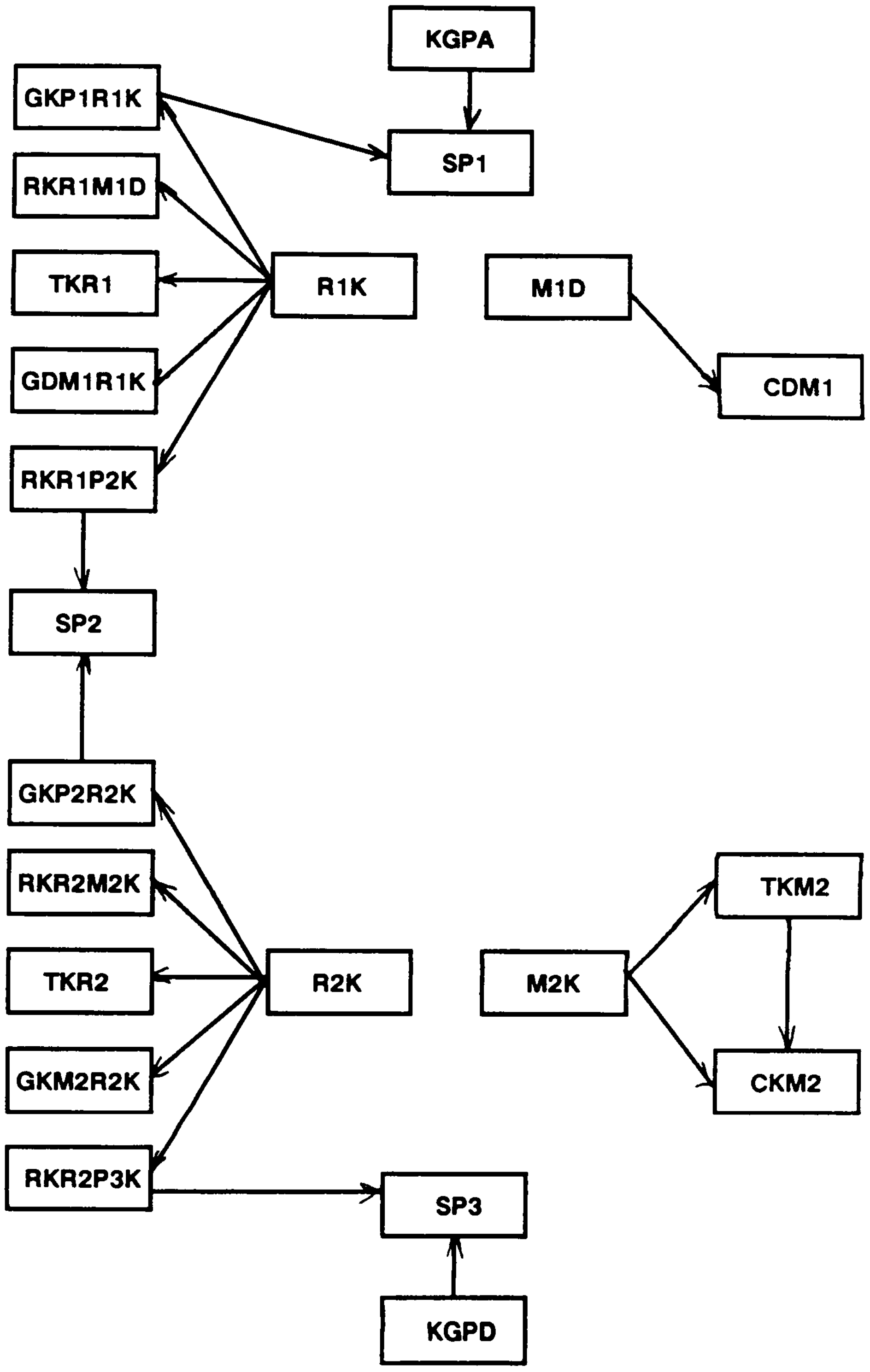
Test Bed Model 13



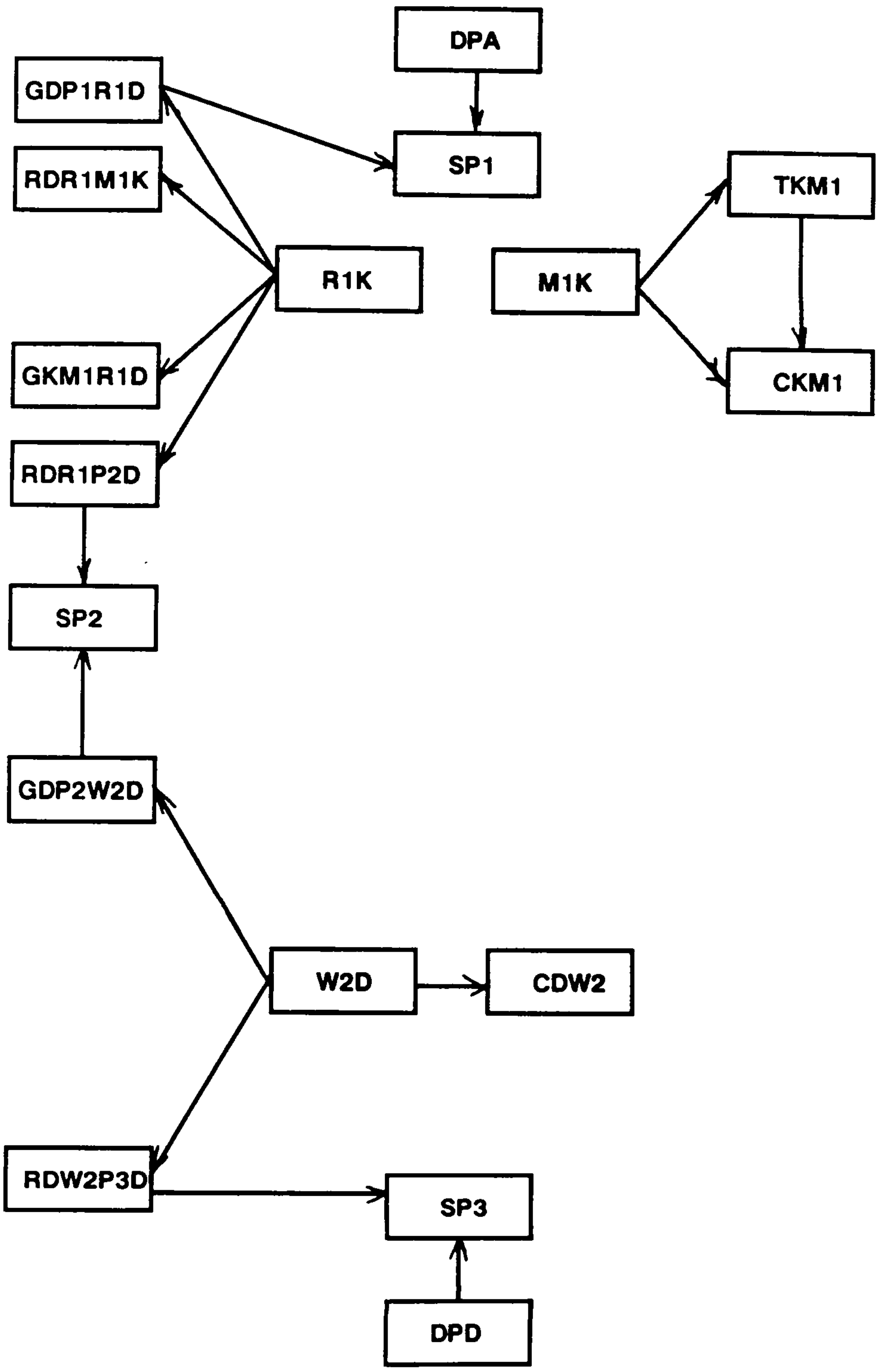
Test Bed Model 14



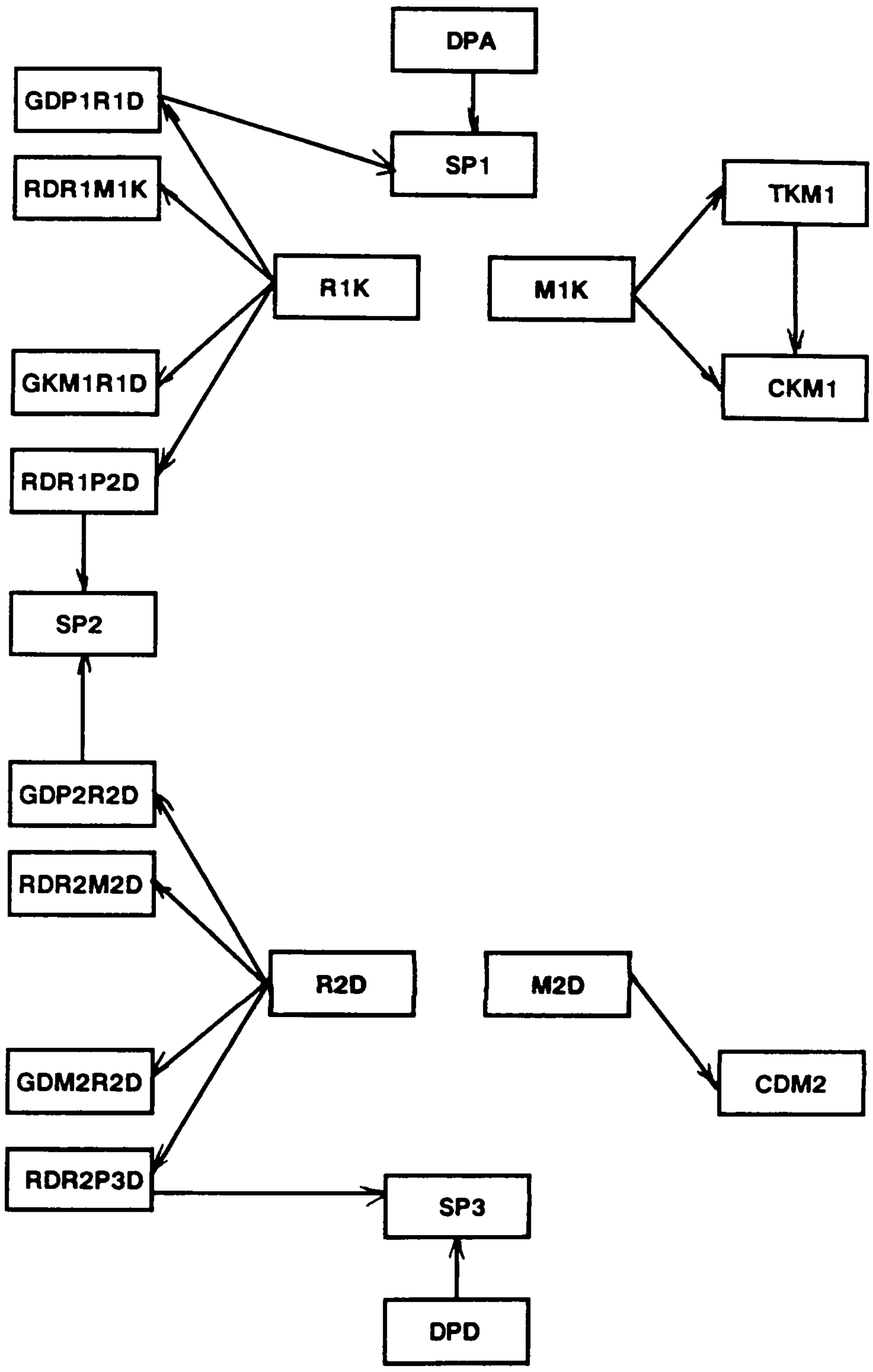
Test Bed Model 15



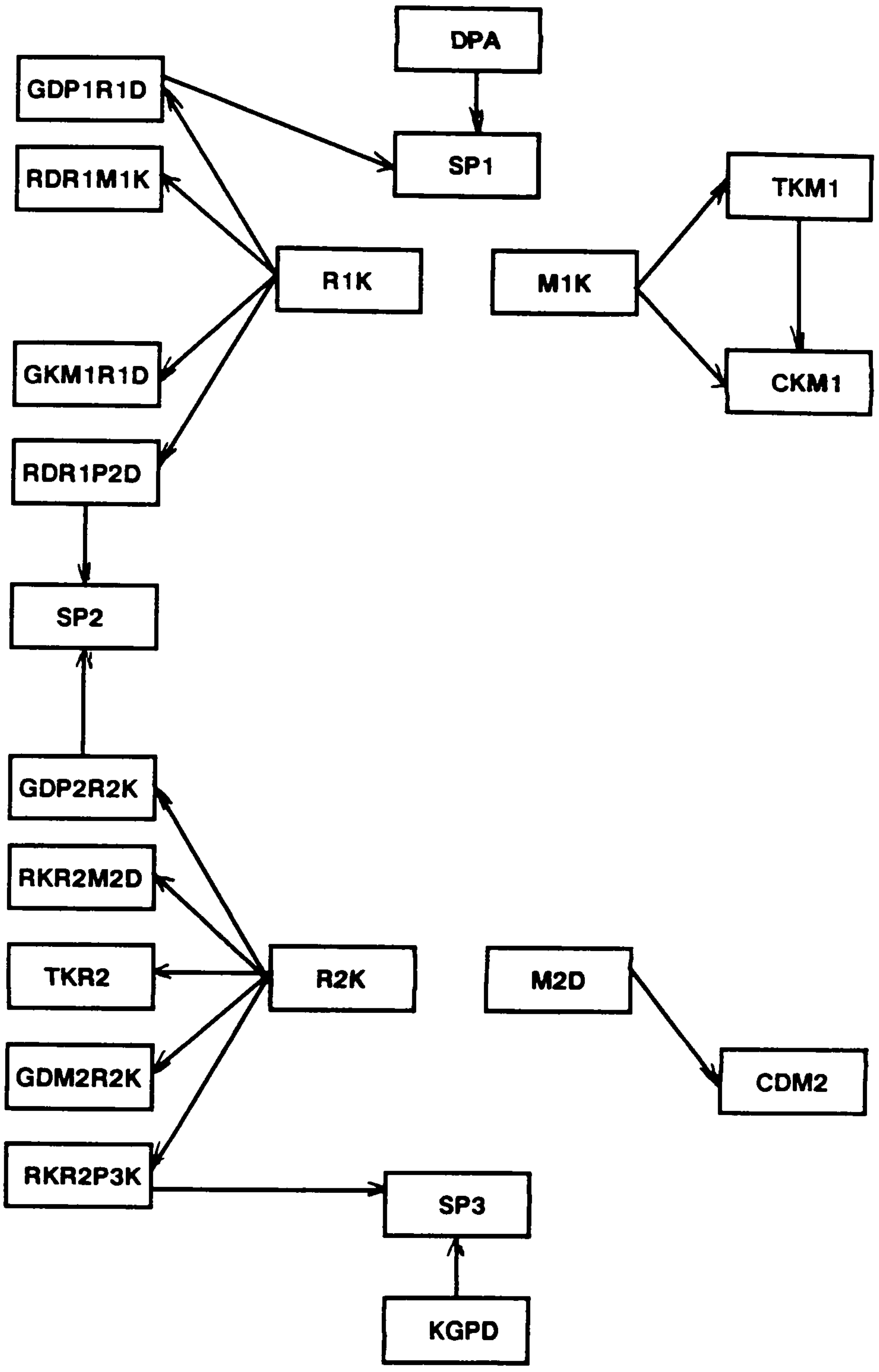
Test Bed Model 16



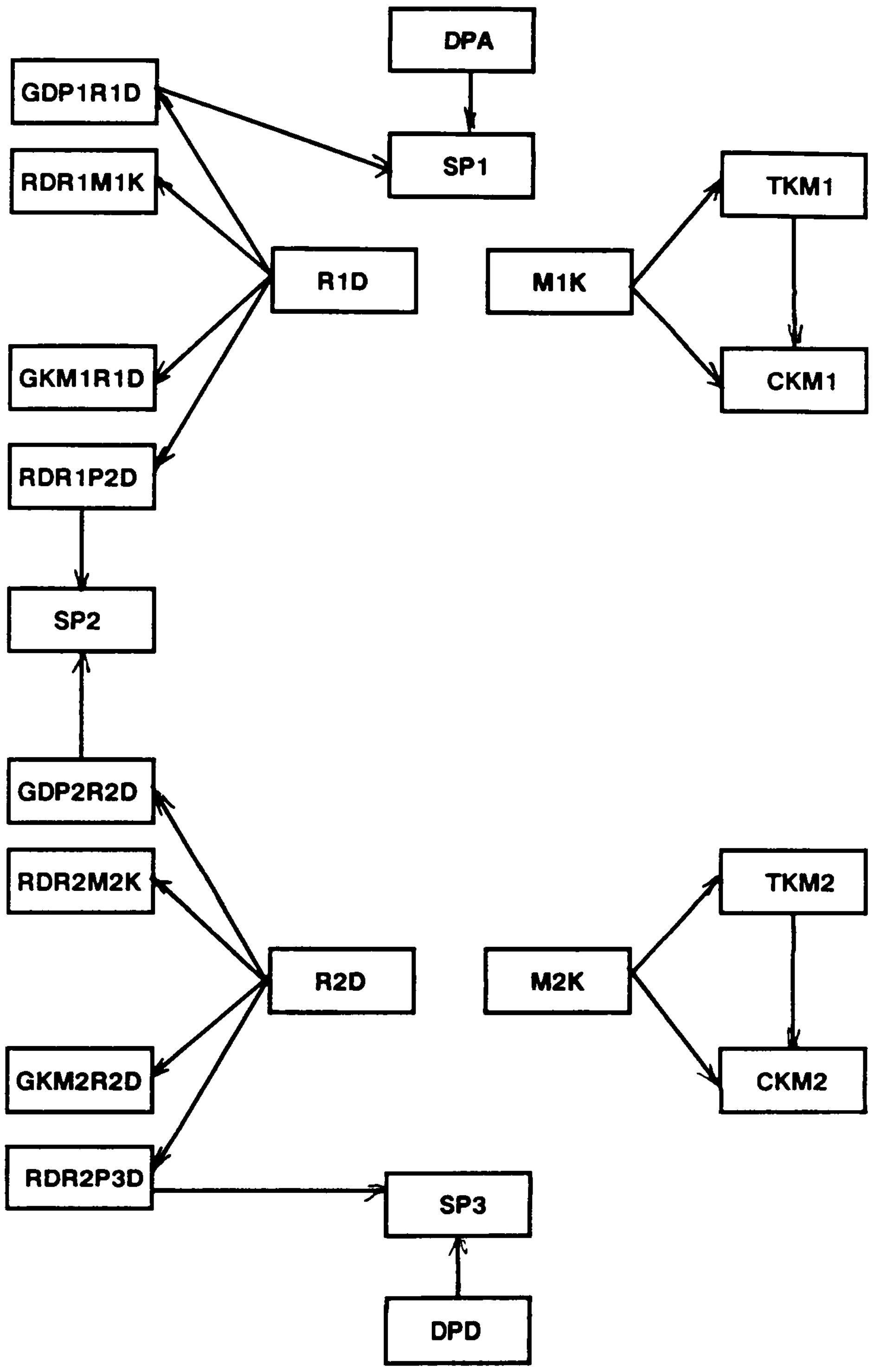
Test Bed Model 17



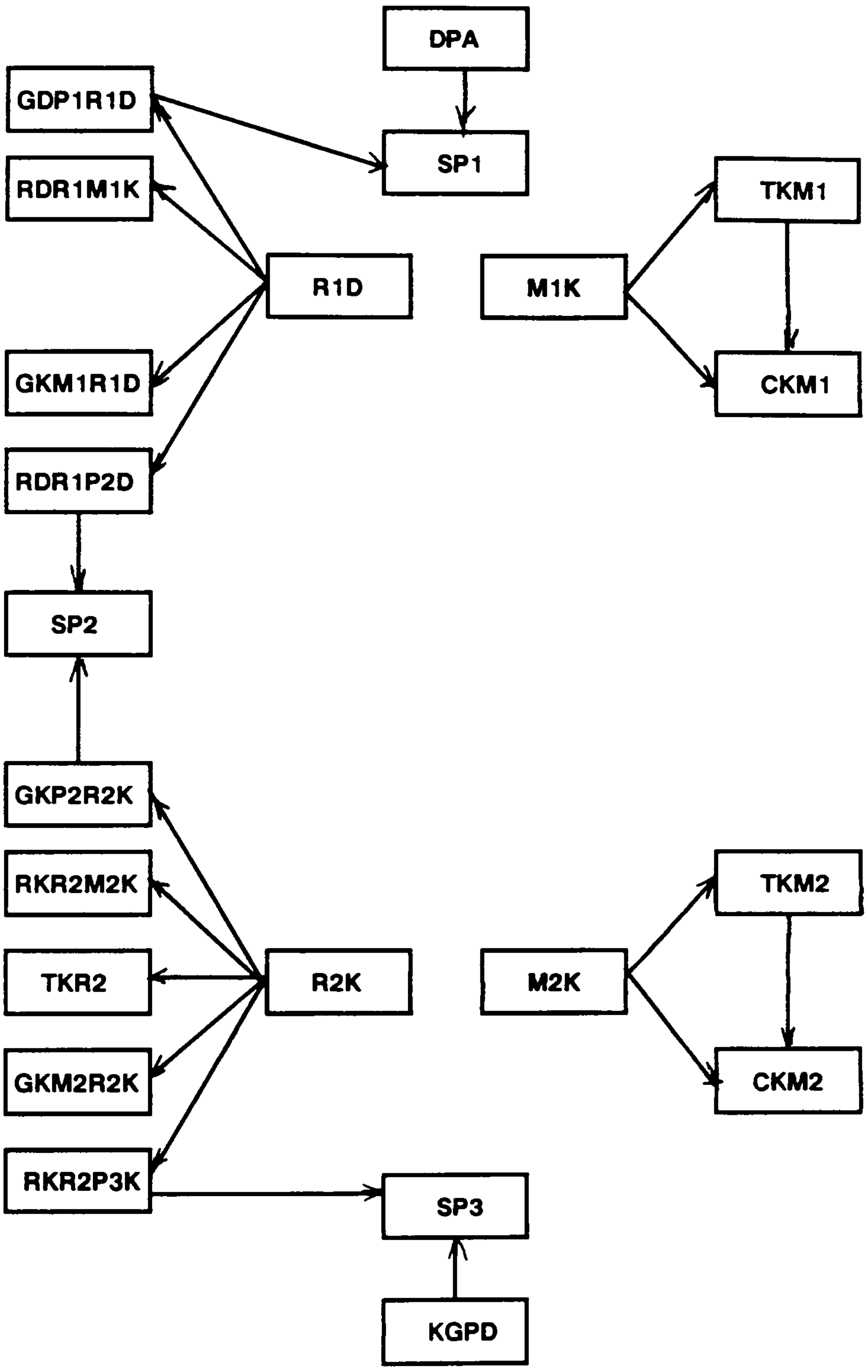
Test Bed Model 18



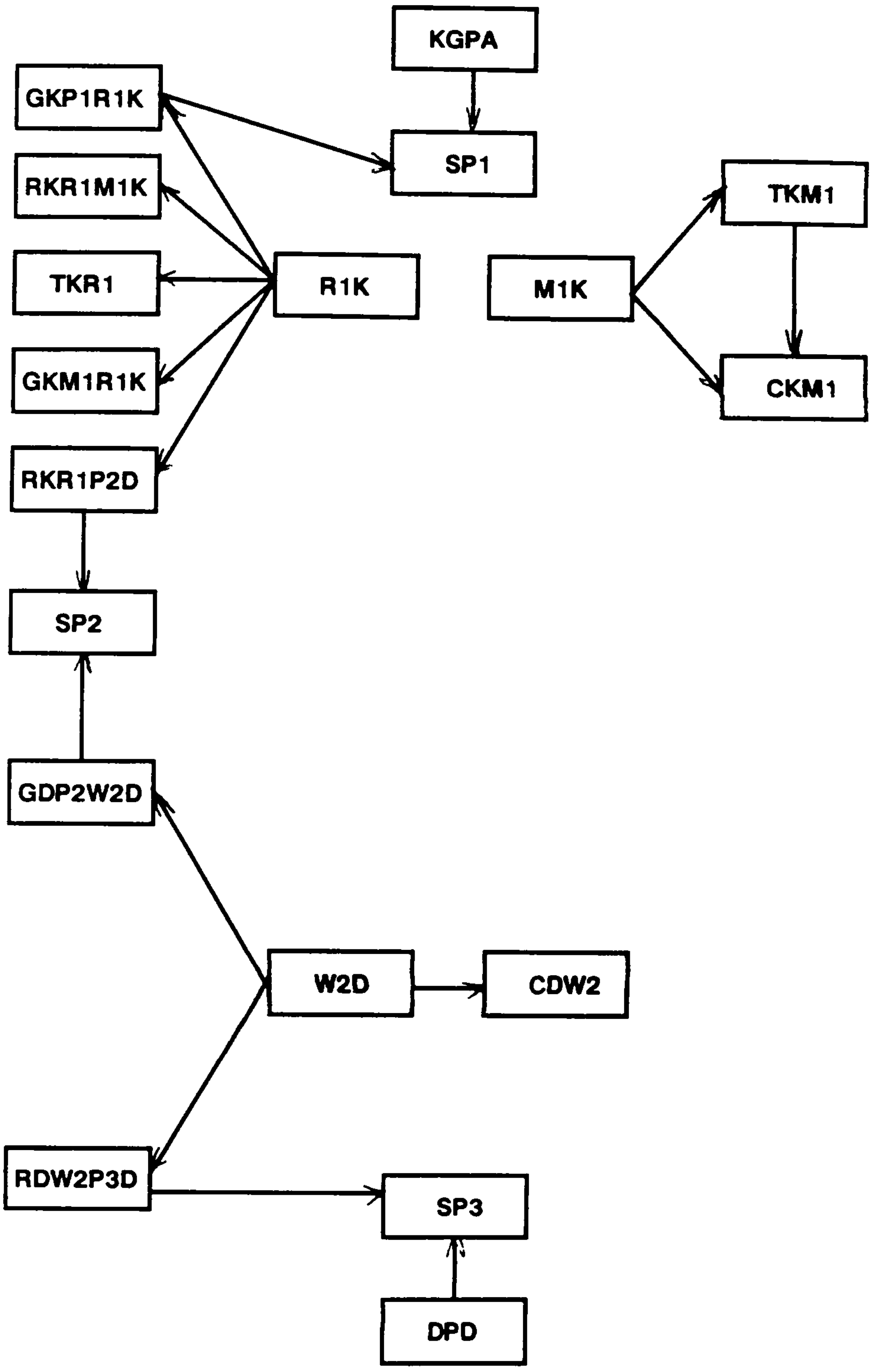
Test Bed Model 19



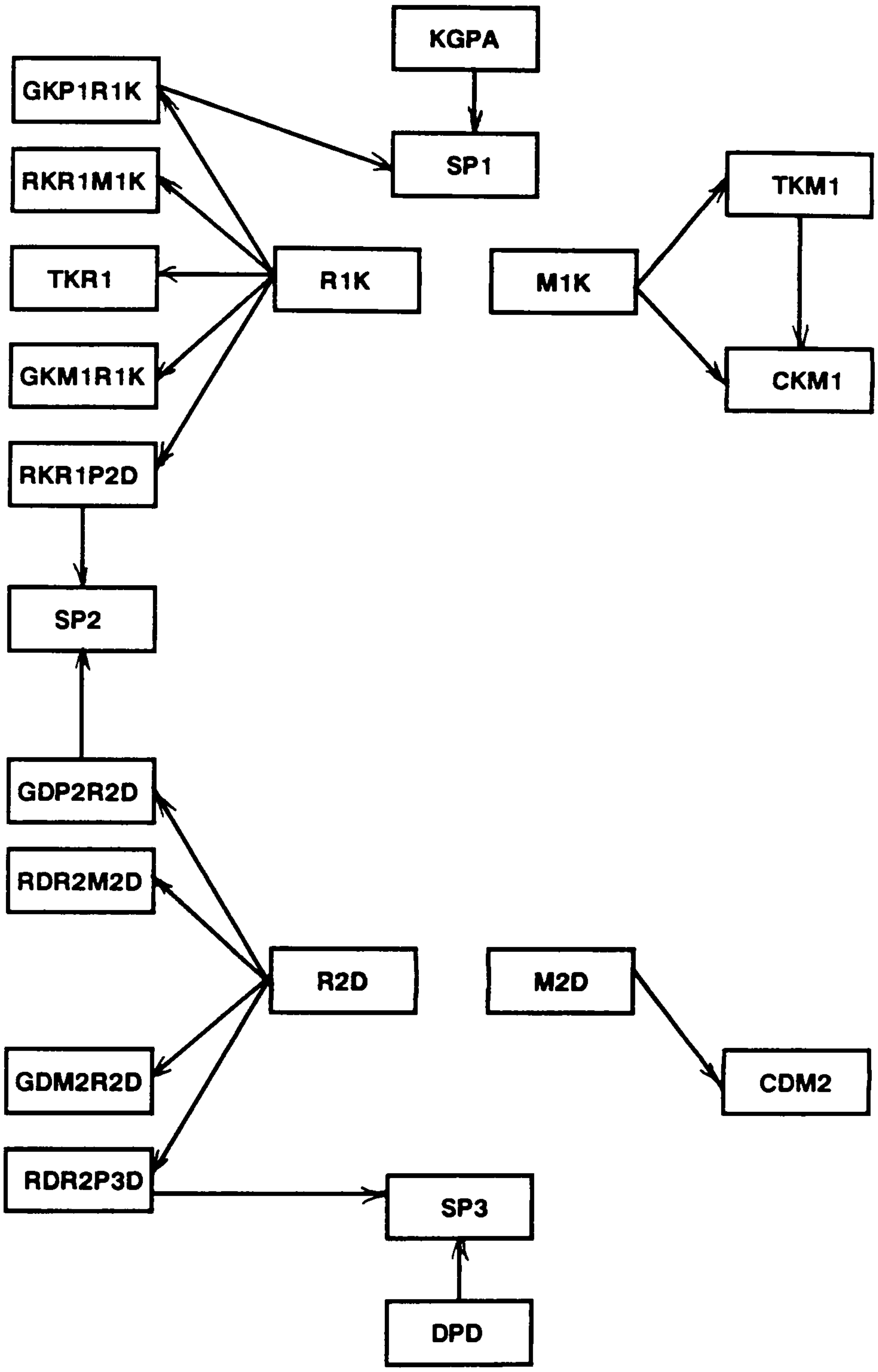
Test Bed Model 20



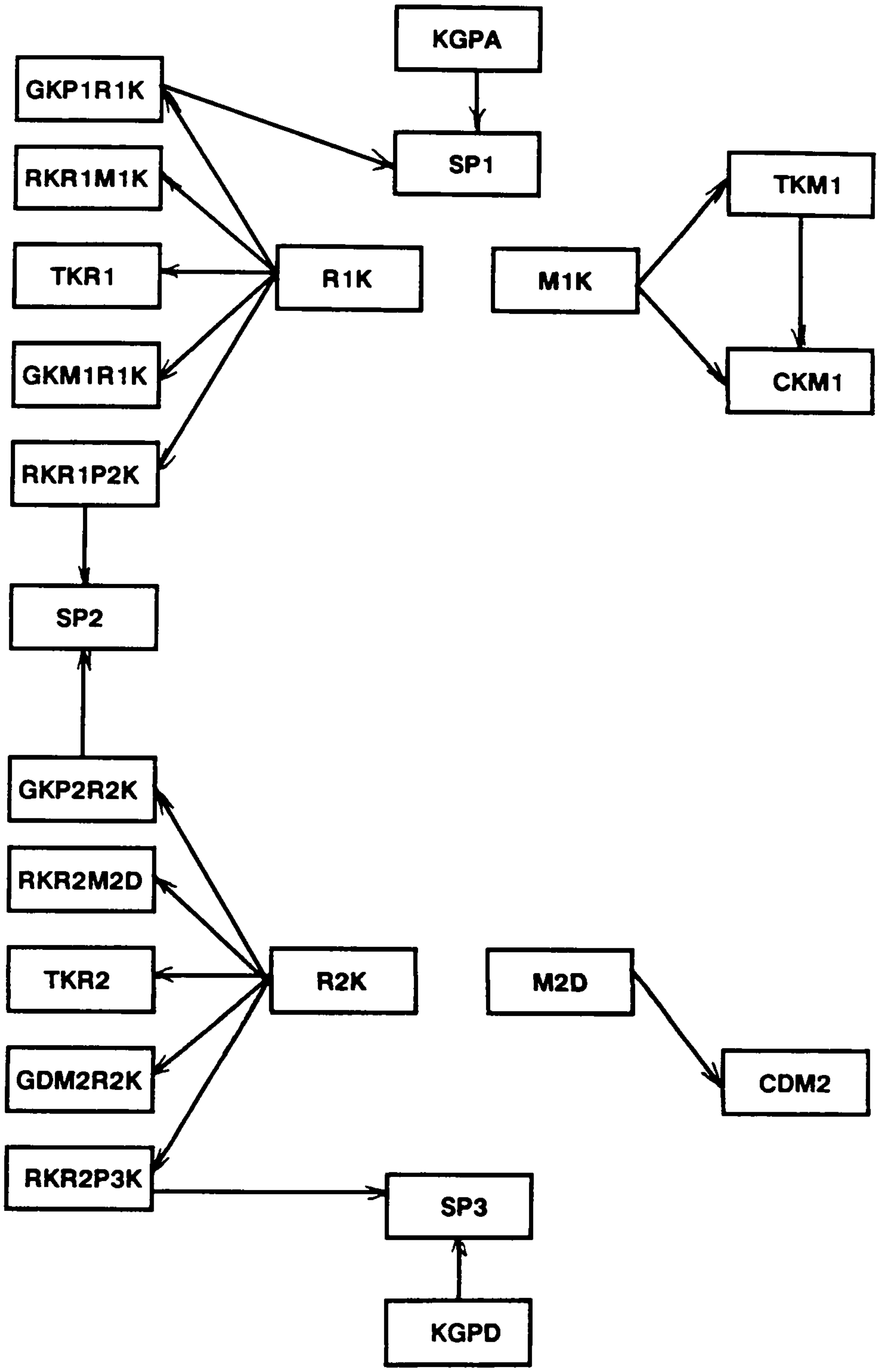
Test Bed Model 21



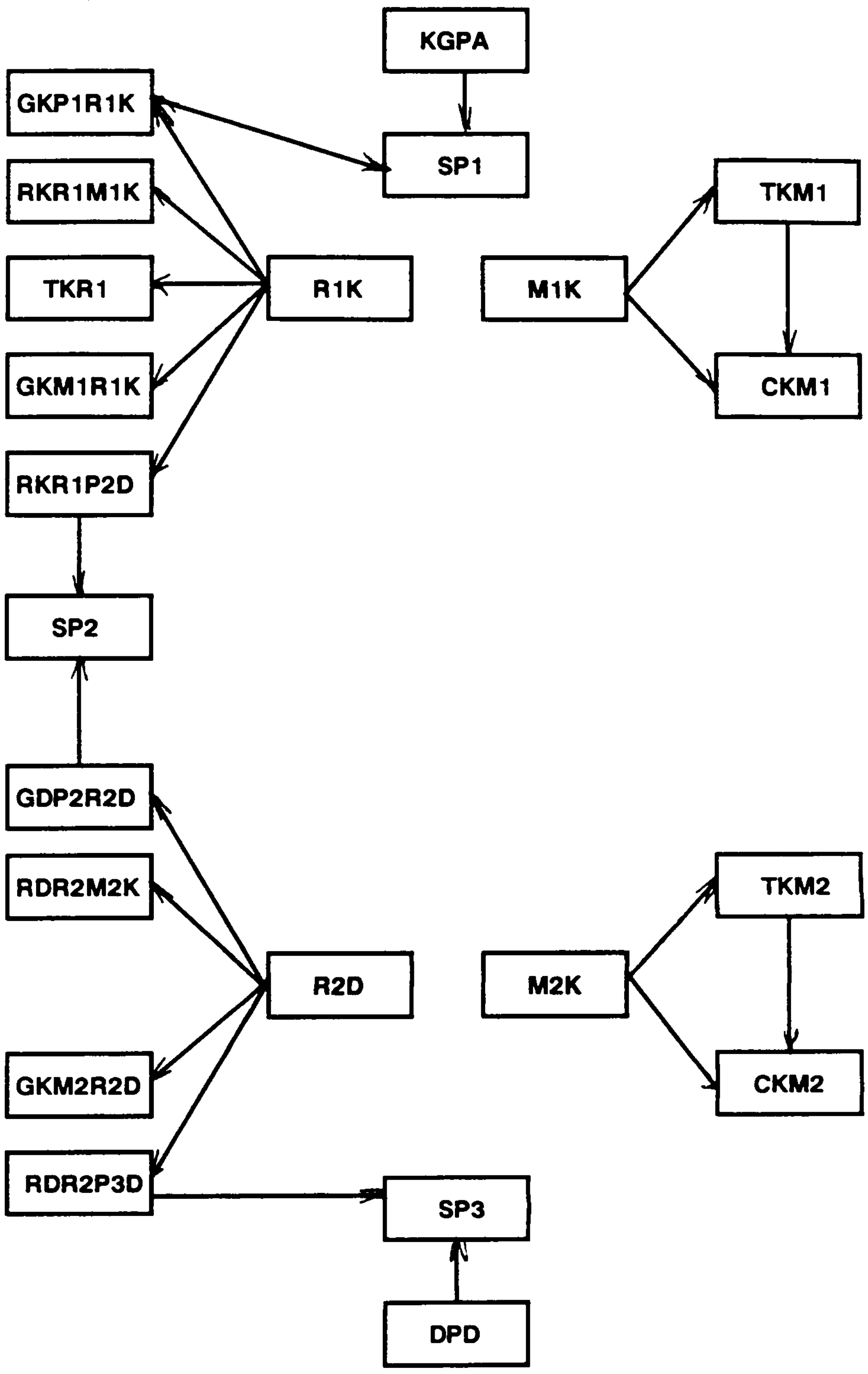
Test Bed Model 22



Test Bed Model 23



Test Bed Model 24



Test Bed Model 25