RSSM/71

Modelling Parallel Systems with Dynamic Structure*

Jack C. Wileden**

COINS Technical Report 78-4

January 1978

Abstract: We present a formal modelling scheme for parallel systems with dynamic structure and investigate the properties of both the scheme and the class of systems which it can be used to represent. Formal semantics are presented for the modelling scheme's constructs and the undecidability of several questions regarding the behavior of modelled systems is demonstrated. A technique for describing the set of possible behaviors for certain systems of this class is presented and related to a subclass of models constructed using the modelling scheme. The potential usefulness of the modelling scheme, particularly as an aid in the design and analysis of dynamically-structured concurrent software systems, is also considered.

*This report reproduces a dissertation of the same title submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer and Communication Sciences) in the Horace H. Rackham School of Graduate Studies at the University of Michigan. Partial support for this research was provided by grants from the Horace H. Rackham School of Graduate Studies and from Sycor, Inc.

**Author's Current Address:
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

ABSTRACT

MODELLING PARALLEL SYSTEMS
WITH DYNAMIC STRUCTURE

by

Jack Craig Wileden

Co-chairmen: William E. Riddle, William C. Rounds

In this dissertation we present a formal modelling scheme for parallel systems with dynamic structure and investigate the properties of both the scheme and the class of systems which it can be used to represent. The modelling scheme was developed for use in the design and analysis of dynamically-structured, concurrent software systems, but has been defined in sufficient generality to permit its use in studying any parallel system which can be characterized as having dynamic structure.

A parallel system with dynamic structure (PSDS) is a system in which multiple activities can occur simultaneously and some or all of whose components may come into and/or go out of existence during the life of the system. Such an organization is frequently found in complex, concurrent software systems. However, none of the formal modelling schemes for parallel systems which have previously been defined allows for the representation of dynamic structure

in other than special cases or particular contexts.

This dissertation defines a formal modelling scheme, the Dynamic Process Modelling Scheme (DPMS), capable of describing parallel systems with dynamic structure in general. The scheme is based upon an abstract programming language, called DYMOL, and a technique for representing the changing configurations of a dynamically-structured system. Formal semantics for DYMOL link the modification of a system's configuration over time to the activities of system components. Examples in the dissertation illustrate the potential usefulness of the modelling scheme in designing and analyzing dynamically-structured, concurrent software systems.

The formal foundation of the modelling scheme allows its use in studying the properties of the general class of parallel systems with dynamic structure. Two different constructions of universal computation devices using the Dynamic Process Modelling Scheme are presented in the dissertation. These constructions are used in proving the undecidability of several questions which might be posed regarding the possible behavior of a general PSDS as modelled using DPMS.

A technique for describing the complete set of possible behaviors for certain systems of this class is also

presented in the dissertation. The technique, called constrained expressions, uses a notation similar to that of regular expressions to give a finite representation of the potentially infinite set of possible behaviors of a PSDS. A PSDS subclass, the class of parallel systems with dynamic connectivity, is defined and an algorithm for deriving a constrained expression from a DPMS model of any system of this subclass is presented.

The dissertation concludes with an examination of the expected usefulness of the Dynamic Process Modelling Scheme, particularly with respect to the design and analysis of dynamically-structured, concurrent software systems.

MODELLING PARALLEL SYSTEMS

WITH DYNAMIC STRUCTURE

by
Jack Craig Wileden

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer and Communication Sciences)
in The University of Michigan
1978

Doctoral Committee:

Associate Professor William E. Riddle,
University of Colorado, Co-chairman
Associate Professor William C. Rounds, Co-chairman
Associate Professor Larry K. Flanigan
Professor Keki B. Irani

This dissertation is dedicated

To Paul, Barbara, Mary, Carolyn and Clifford,
sources of lifelong encouragement and support;

To Carolyn P. Steinhaus,
a uniquely fine and special person;

And to the memory of
Theresa Motter Petty and
Dr. Lewis Alison Wileden

ACKNOWLEDGEMENTS

The inspiration, motivation and guidance leading to the completion of this dissertation have come primarily from Professor William E. Riddle. His direction, encouragement and support have always been generously given and are most gratefully appreciated. His teaching skills and research insights have provided a model worthy of emulation. I consider myself privileged to have known Bill Riddle as teacher, advisor, research director, exemplar and friend.

It is my pleasure to acknowledge Professor William C. Rounds, who has directed the final arduous phases of this project. His incisive and rigorous thinking has benefited me frequently, helping to clarify my understanding of many of the issues addressed in this dissertation.

I also wish to express my sincere appreciation to the other members of my doctoral committee, Professors Larry K. Flanigan and Keki B. Irani. Their thoughtful criticisms and guidance have contributed significantly to the successful completion of my graduate studies.

I am, as always, grateful to my first and best English teacher, Barbara Petty Wileden. Her excellent, short-order proofreading of this dissertation is but the latest of the innumerable things for which I can never hope to adequately thank her.

So many others, too numerous to mention here, have contributed in some way to the completion of this undertaking. I can only trust that all of them -- faculty, fellow graduate students and friends -- are aware of my deep gratitude for everything they have done.

Finally, I wish to thank both Sycor, Inc. and the Horace H. Rackham School of Graduate Studies for providing partial support at various stages in the research and writing of this dissertation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER I

INTRODUCTION

Contemporary complex software systems are frequently organized as collections of interacting concurrent processes in which the processes may be created or destroyed and patterns of interprocess communication may change over the course of system operation. Such software systems are representatives of the general class of parallel systems with dynamic structure. Formal models of such systems could be useful in studying their properties and could serve as a basis for design and analysis methods applicable to complex software systems with dynamic structure. However, existing models of parallel computation either assume a static system structure or have only very specialized and limited provisions for handling dynamic structure. In this dissertation we develop a new formal modelling scheme capable of representing parallel systems with dynamic structure and illustrate its use in designing and analyzing such systems. The scheme is also utilized in establishing several undecidability results regarding the behavior of systems in this class. In addition, a behavioral description technique applicable to such systems is defined

here and related to certain types of models described using the new scheme.

Parallel Systems with Dynamic Structure

The concept of parallel systems with dynamic structure and the formal modelling scheme developed herein for representing such systems were both motivated by an interest in the design of complex software systems. Their application is, however, by no means restricted to the domain of software nor even computing systems. Rather, we may make the following general definition of a parallel system[1]:

> Definition I.1: A parallel system is any system consisting of a collection of components which may operate concurrently, such that the overall behavior of the system results from the coordinated activities of the individual components, with the coordination achieved through interaction among the components.

This definition may in turn serve as the basis for a general definition of a parallel system with dynamic structure, as follows:

> Definition I.2: A parallel system with dynamic structure (PSDS) is any parallel system in which components may be created and destroyed and in which intercomponent interaction paths may be established and closed during the course of the system's operation.

Under these general definitions, numerous examples of

---

[1]In this dissertation we use the terms 'concurrent' and 'parallel' interchangeably, making no distinction between actual and conceptual simultaneity.

parallel systems with dynamic structure may be found outside the realm of computer systems. The modelling scheme developed here could, therefore, potentially be employed in the investigation of these systems.

As one example of a parallel system with dynamic structure found in nature, consider a hive of honey bees. The individual bees may be viewed as components of the parallel system which is the hive itself, their activities coordinated to produce the behavior of the overall system through an elaborate information communication behavior, called a dance by Frisch [Fris71]. Components of the system are created and destroyed during system operation due to the birth and death of individual bees. Interaction paths are established and closed as bees arrive at and depart from the honey comb on which the dance takes place. The honey bee hive thus exhibits all the attributes of a parallel system with dynamic structure as set forth in the above definition.

While many additional instances of parallel systems with dynamic structure could be cited which are unrelated to computer systems, our basic interest is in dynamically-structured concurrent software systems. In the remainder of this dissertation we shall restrict our attention and examples to such software systems and adopt terminology appropriate to them. In particular, the components of parallel software systems are generally known as processes

[Horn73], while the parallel systems themselves may be referred to as systems of cooperating sequential processes [Dijk68] or systems of interacting parallel processes. These two phrases are synonymous but emphasize different aspects of system organization; the former stressing the sequential nature of the activity of the individual processes whose cooperation results in overall system behavior, while the latter underscores the fact that the component processes are operating in parallel, with their interaction determining the system's performance.

Among existing software systems, examples of parallel systems with dynamic structure are found chiefly in the areas of operating systems and artificial intelligence problem solving systems. The HYDRA multiprocessor operating system kernel, developed at Carnegie-Mellon University [Wulf74], and the nucleus of the multiprogramming system for the Regnecentralen RC4000 computer [Brin70] both support parallel systems with dynamic structure. These systems are based upon the creation and destruction of processes and the modification of interprocess communication paths during the course of system operation. Such artificial intelligence problem solvers as the HEARSAY natural language understanding system [Penn77] and the VISIONS system for visual scene analysis [Hans76] are also organized as parallel systems with dynamic structure. The creation of various hypothesis-testing processes, called 'knowledge

sources', is fundamental to the operation of the techniques employed for problem solution in the HEARSAY and VISIONS systems. The goals of facilitating the understanding of complex software systems like those discussed here, and of contributing to the development of design and analysis methods for such systems, were fundamental to the development of the modelling scheme presented in this dissertation.

Software Reliability and Formal Models

Formal models can serve as very useful tools for clarifying the operation of software systems. A formal model provides an abstract representation which focuses on only the particular features of a software system pertinent to understanding some aspect of its operation and allows for the derivation or verification of system properties through well-defined formal manipulations. As a result, formal models provide a foundation for much of the work done by computer scientists in the area of software reliability.

This is quite evident in the areas of design and analysis of sequential computer programs where a great deal of software reliability work has been done. Design techniques employing the concepts of structured programming and stepwise refinement, advocated by Dijkstra [Dijk72], Wirth [Wirt71] and others, are based on the formal work of

Bohm and Jacopini [Bohm66]. The "proof of program correctness" approaches to program verification and analysis advanced by Manna [Mann74], Floyd [Floy67] and others are similarly founded in the formal theory of logic and the program schemata model developed by Ianov [Rutl64].

In the area of concurrent software systems neither design nor analysis techniques are as fully developed as they are in the case of sequential computer programs. Nevertheless, formal models of parallel computation have made significant contributions to our basic understanding of concurrent programming in general and interacting parallel processes in particular. They have also been fundamental to several of the design and analysis techniques which have been proposed for concurrent software systems. Campbell's path expression notation for specifying synchronization constraints has been described and analyzed using Petri nets [Camp76]. Keller defines a formal model as the basis for his proposed verification method for parallel programs [Kell76]. Riddle's modelling scheme has been used as a basis for a concurrent software system design method [Ridd73] and has applications to analysis of concurrent software systems [Wile76]. It has also provided the foundation for an automated design aid for concurrent software systems [Ridd77a]. These examples suggest that the development of appropriate formal models may be a significant additional step toward the eventual development

of design and analysis techniques for complex, concurrent software systems.

Existing Formal Models of Parallel Computation

A number of formal models of parallel computation have been advanced and studied by various computer scientists[1]. But while several of these existing models have contributed ideas potentially valuable to the study of parallel systems with dynamic structure, none of them is actually appropriate to the modelling of this class of parallel systems.

C. A. Petri developed the first modelling scheme for parallel computation [Petr66]. A Petri net consists of a bipartite directed graph [Hara69], whose vertices are known as places and transitions, and a set of rules governing the movement of tokens from place to place within the graph. Under specified enabling conditions, a transition may 'fire', removing tokens from all the places adjacent to the transition and placing tokens in all the places adjacent from the transition (where 'adjacent to' and 'adjacent from' have their technical digraph theory meaning [Hara69]). Petri's definition allows the enabling conditions to hold for several transitions in the net simultaneously and thus transition firing activity may occur in parallel. Several

---

[1]Surveys of these modelling schemes may be found in [Bred70], [Mill73], [Pete74a] and [Pete74b].

other models based on minor modifications to Petri's have appeared, including UCLA graphs ([Baer70], [Cerf72]), James Peterson's p-nets [Pete74a], colored Petri nets [Zerv77] and the model upon which Keller based his parallel program verification techniques [Kell76]. Extensive research has also been done on the properties of Petri nets [Hack76a] and on analysis of Petri nets in terms of vector addition systems [Hack75] and Petri net languages [Hack76b]. But although Petri's modelling scheme is very general and applies to a broad range of parallel systems, it cannot model parallel systems with dynamic structure in any natural way, since a Petri net has a fixed structure and no provision for altering that structure.

The parallel computation graph modelling scheme due to Karp and Miller [Karp66] consists of a directed graph in which the vertices represent computational operations and the edges are taken to be data queues. Under specified enabling conditions, a vertex may remove one datum from each of its inbound edges and place a result datum on each of its outbound edges. As with Petri nets, enabling conditions may arise simultaneously at several vertices in a parallel computation graph and hence parallel activity may occur. Karp and Miller were able to prove several properties regarding determinacy of computation, termination of execution and bounds on data queue length with respect to this model. A similar model developed by Duane Adams

[Adam68] permits more sophisticated execution rules for the vertices and provides for a limited amount of implicit dynamic restructuring of the graph through the definition of procedure vertices. Initiation of execution in a procedure vertex causes the vertex to be replaced by a graph detailing the procedure's operation (similar to macro expansion in a programming language), thus providing for a limited restructuring of the Adams graph and for the modelling of recursion, an unusual attribute for a model of parallel computation. Despite the limited restructuring available in Adams graphs, neither the Karp and Miller nor the Adams formulation is appropriate to the modelling of parallel systems with dynamic structure since each scheme is based on an essentially fixed graphical structure with no general capability for explicitly altering that structure.

The parallel program schemata modelling scheme, also due to Karp and Miller ([Karp67], [Karp69]), is automata theoretic in nature. The initiation and termination of operations in a parallel program schema are represented by state transitions, with the current state of the schema indicating exactly which operations or processes are currently in execution. Parallel activity is then modelled by states which represent the simultaneous execution of several operations. Parallel program schemata provide a reasonably general model for parallel computation, and investigation of their properties led to Karp and Miller's

development of the very useful vector addition system formalism. However, due to the necessity of supplying an exhaustive (fixed) state set and complete (fixed) state transition function as part of any parallel program schema, this modelling scheme does not naturally allow for the modelling of parallel systems with dynamic structure.

Riddle has developed a modelling scheme for complex, concurrent software systems [Ridd76] which includes a technique for behavioral specification. The Program Process Modelling Language (PPML) is a means for modelling parallel computation based on a graphical representation of process interconnection and an abstract programming language description of each process' operation. Message Transfer Expressions (MTEs) provide a separate, formal language-based behavioral description for the scheme. PPML and MTEs focus on process interaction as represented by interprocess message transmission. An effective procedure has been defined for deriving an MTE from a PPML description, after which a limited amount of behavioral analysis of the model may be performed using the MTE. However, since it is based on a fixed process intercommunication structure, this scheme too is inadequate for modelling parallel systems with dynamic structure.

Two recently proposed modelling techniques for studying issues related to protection in operating systems [Jone73]

allow for dynamic structure but have no provision for representing parallel computation. One of these schemes, due to Harrison, Ruzzo and Ullman [Harr76], is based upon a dynamically-structured access matrix whose entries are sets of 'generic rights'. The matrix can be manipulated by a set of commands which may alter its contents and add or delete rows and columns. This modelling scheme has been used to represent various operating system protection mechanisms and to prove the undecidability of certain protection system problems. The second scheme, proposed by Lipton and Snyder [Lipt77], employs a directed labelled graph and a set of rewriting rules which can alter the structure of the graph by adding or deleting vertices and edges. The decidability of some protection systems questions in certain specific situations has been demonstrated using this modelling scheme. Although both formulations are capable of representing dynamic structure and thus bear some resemblance to the modelling scheme presented in this dissertation, neither makes any provision for describing parallel activity. The notion of processes, as active components or loci of control which issue commands or invoke rewriting rules, is absent from both schemes. Thus, while appropriate to their intentionally limited role as aids in the investigation of protection systems, these modelling schemes are inadequate for the representation of parallel systems with dynamic structure.

Outline of the Dissertation

The remainder of this dissertation presents a modelling scheme for parallel systems with dynamic structure and illustrates the use of this scheme in investigating, designing and analyzing such systems. Chapter II offers an overview of the modelling scheme, introducing its major features and underlying assumptions. In Chapter III we discuss the abstract programming language which is the foundation of the scheme's technique for the description of processes. Chapter IV presents the complete modelling scheme and illustrates its use through an example which demonstrates the scheme's potential as a basis for design and analysis techniques applicable to concurrent software systems with dynamic structure. In Chapter V we consider the computational power of parallel systems with dynamic structure, prove that certain questions regarding these systems are undecidable in general, and discuss the implications of this result. In the next chapter we investigate a subclass of the class of parallel systems with dynamic structure which corresponds to a situation of interest in the design of complex, concurrent software systems. Specifically, Chapter VI defines parallel systems with dynamic connectivity and describes their representation in the modelling scheme, then presents a formal language-based description technique for certain aspects of the behavior of such systems and an algorithm for deriving these

behavioral descriptions from models of the systems. Chapter VII concludes the disseration, summarizing its contents, suggesting extensions and directions for future research, and examining the potential utility of the modelling scheme for parallel systems with dynamic structure.

CHAPTER II

OVERVIEW OF A MODELLING SCHEME

This chapter provides an introduction to the Dynamic Process Modelling Scheme (DPMS), a formal modelling technique developed for use in studying parallel systems with dynamic structure. The intention here is to explain the viewpoint underlying DPMS, motivate the use of the various features which constitute the modelling scheme and offer a high-level, unified description of the complete Dynamic Process Modelling Scheme. Chapters III and IV present a fully detailed development of DPMS -- the present chapter overviews the structure of and the rationale behind the modelling scheme in preparation for that more elaborate exposition.

The DPMS Viewpoint

As we have indicated in the introduction, a wide range of phenomena may be characterized as parallel systems with dynamic structure and hence could be modelled using the Dynamic Process Modelling Scheme. However, the scheme was developed primarily for use in studying complex, concurrent

software systems with dynamic structure and this intended utilization has influenced several aspects of DPMS. In particular, it has impacted the choice of basic representational units for the scheme, its representational focus, its means of representing communication and coordination within the modelled system, and its representation of parallel activity.

A standard approach for facilitating the understanding of any complex system, suggested by Simon [Simo62], is to decompose the system into an organized collection of smaller, simpler parts. Represented in this manner, the system can then, in principle, be understood in terms of the activity of these smaller, simpler parts and the interactions among them. In the specific case of complex, concurrent software systems this approach translates into decomposition into processes [Horn73] or modules [Parn72] whose individual internal activity is taken to be purely sequential. The result may be referred to as a system of cooperating sequential processes [Dijk68] or, equivalently, a system of interacting parallel processes, where concurrency may be an attribute of a group of processes (which can operate in parallel) but not of any individual process (whose activity is strictly sequential).

This standard view of complex software system structure has been adopted in the Dynamic Process Modelling Scheme.

As its name suggests, the scheme is process-based, i.e. the components or basic DPMS representational units for describing parallel systems with dynamic structure are processes. The internal activity of each individual DPMS process is taken to be sequential and thus parallel activity can only be an attribute of a collection of processes in a DPMS model.

This process-based view is not necessarily a limitation on the descriptive power of the Dynamic Process Modelling Scheme. As mentioned above, the viewpoint is one widely accepted in the area of complex, concurrent software systems, our foremost concern in evaluating the usefulness of the modelling scheme. Moreover, numerous other systems can be meaningfully modelled in these terms; the bee hive system discussed in the previous chapter can, for example, be reasonably modelled by representing each individual honey bee as a process. Furthermore, in those instances where the individual components of a parallel system with dynamic structure are most naturally viewed as having internal parallel activity, they may be represented as subcollections of processes in DPMS. In general, representing a parallel system with dynamic structure as a collection of cooperating sequential processes is simply a matter of finding an appropriate level of system decomposition. Therefore, the fact that the Dynamic Process Modelling Scheme is process-based can be expected to have little or no negative impact

on the scheme's descriptive power while it renders the scheme a very natural one for use in studying complex, concurrent software systems with dynamic structure.

The decomposition of a complex system into an organized collection of smaller, simpler parts as advocated by Simon is indeed a useful step toward understanding the system, but it by no means completely solves the problem. While the parts do in fact become simpler, the interactions among them typically remain quite difficult to comprehend. This generalization is certainly borne out in the case of the decomposition of a complex, concurrent software system into a system of interacting parallel processes. The behavior of individual sequential processes is reasonably straightforward to understand, and several formal techniques have been advanced for verifying that sequential process behavior adheres to specified criteria for correctness ([Mann74], [Hoar69], [Floy67]). Understanding the overall behavior resulting from their interaction remains a significant problem, however. This situation is aptly reflected by the publication of incorrect solutions to comparatively simple concurrent software problems ([Hyma66], [Pati71]) and the absence of any widely-accepted techniques for verifying the correctness of such systems[1]. Quite evidently, comprehension problems are only compounded in a

---

[1]Examples of possibilities which have been put forth include [Laue72], [Owic75], [Kell76] and [Ridd76].

system which does not merely exhibit concurrency but is in fact a parallel system with dynamic structure.

Consideration of the relative complexity of comprehension connected with various aspects of complex systems in general, and concurrent software systems with dynamic structure in particular, has influenced the representational focus of the Dynamic Process Modelling Scheme. The scheme is distinctly oriented toward the description and analysis of the interaction among processes within a modelled system. In the case of parallel systems with dynamic structure this interaction involves not only interprocess communication but process creation and destruction and the modification of interprocess communication paths as well. All these types of process interaction can be explicitly represented in DPMS. Details of internal process computation, on the other hand, are not explicitly representable in DPMS process descriptions but may only be abstractly represented in terms of their impact on process interaction. Nondeterminism plays an important role in this abstraction of internal process computation, allowing for the succinct description of all possible sequences of significant (i.e. interaction-related) process activities which could occur due to various possible outcomes for the unelaborated internal computational activity of the process. This representational focus of the Dynamic Process Modelling Scheme, emphasizing process

interaction while abstracting internal process computation, makes DPMS ideally suited for use in studying the most complex and challenging aspects of the behavior of parallel systems with dynamic structure.

The DPMS mechanisms for representing communication and coordination among system components have similarily been chosen largely due to their applicability to the description of complex, concurrent software systems. This interprocess communication and coordination is based upon the transmission of messages from one process to another. Message-based communication is commonly found in concurrent software systems (e.g. [Brin70], [Feld77]) and has previously been employed in modelling schemes ([Ridd76], [Zave77]) and a design tool ([Ridd77a]) for complex, concurrent software systems. In the Dynamic Process Modelling Scheme, communication and coordination among processes are depicted by the transmission of messages which are representatives of a finite number of distinguishable equivalence classes, called message types, along communication pathways known as channels. The necessary mediation of message transmission activity among asynchronous processes, i.e. the storage of messages which one process has sent but no process has yet requested and the eventual satisfaction of requests for messages which could not be fulfilled at the time they were lodged, is represented in DPMS by special purpose components called

links. Links provide an unbounded, unordered storage facility for messages while continually monitoring the channels to which they are connected for any outstanding requests for messages. They are thus an abstract generalization of the typical message buffering capabilities found in systems of interacting parallel processes, permitting communication and coordination without requiring lock-step synchronization among the processes in the system. Both the message type and link concepts have been adapted from similar notions in Riddle's complex software modelling scheme ([Ridd76]). Together they provide DPMS with a reasonably flexible and general representation for communication and coordination within a parallel system with dynamic structure. At the same time, they are closely related to mechanisms which we would expect to find in a complex, concurrent software system, thus enhancing the applicability of DPMS to the design and analysis of dynamically-structured, concurrent software systems.

Finding an appropriate representation for simultaneous activities in a parallel system has been a problem for modellers of these systems since the inception of work in this area. Petri's original modelling scheme for parallel systems [Petr66] and most of its successors have admitted of the possibility of two or more events actually occurring simultaneously. In general, however, descriptions of the behavior of systems modelled using these schemes do not, in

fact, permit the natural representation of true simultaneity.[1] Rather, parallel operation is expressed as the arbitrary interleaving of sequences of indivisible events. That is, two events x and y which could occur simultaneously in a parallel system are modelled as producing two possible system behaviors, one in which x precedes y and one in which x follows y. This interpretation is applied to every pair of potentially concurrent events in the modelled system with the result that the system's behavior is represented by the set of all possible event sequences resulting from the interleaving, or shuffling, of every set of potentially concurrent events.

This representation of parallel behavior is not as restrictive as it might at first appear and is, in particular, quite acceptable for the description of the operation of concurrent software systems. In instances other than those involving component interaction, the fact that two components in a parallel system are indeed performing their activities simultaneously is essentially irrelevant to understanding the system's operation. Generally, when both are completed it makes little

---

[1] An exception is the parallel program schemata technique of Karp and Miller ([Karp67], [Karp69]) in which certain states represent the simultaneous occurrence of multiple operations. Yet even in this scheme the initiation and termination of operations can only occur individually, not simultaneously, so that simultaneous initiation or termination must be represented by arbitrary interleaving of these events.

difference whether one operation preceded the other or both occurred at the same instant. The outcome of component interaction situations, of course, may crucially depend upon the relative ordering or simultaneity of the operations of the respective components. If true simultaneity is either the only acceptable or the only unacceptable possibility for operation ordering in a given situation, schemes employing the arbitrary interleaving representation will be unable to provide the most useful information on system behavior. However, it is frequently the case that true simultaneity is not the only interesting possibility for operation ordering in a parallel system. Instead, acceptable (or unacceptable) behavior can result from either simultaneous behavior or from the occurrence of the given events in certain particular orders. In such cases, the arbitrary interleaving representation can frequently provide useful information regarding the behavior of the modelled system. For complex, concurrent software systems this representation technique is especially applicable, since process interactions in such systems are generally, at some level, prevented from being truly simultaneous by the inherently sequential nature of the computer hardware which provides the connection between the processes. Moreover, of course, this representational viewpoint is natural to such familiar parallel software situations as a multiprogramming system running on a uniprocessor computing system.

Thus this technique of representing parallel activity as the arbitrary interleaving of indivisible events has been adopted in the Dynamic Process Modelling Scheme as sufficient for and appropriate to a large class of parallel systems with dynamic structure, in particular those dynamically-structured, concurrent software systems which are our primary concern.

Overview of DPMS

The Dynamic Process Modelling Scheme is a formal modelling technique intended for use in studying parallel systems with dynamic structure. Due in part to our particular interest in the investigation of dynamically-structured, concurrent software systems as discussed above, DPMS is a process-based scheme which focuses upon representing dynamic structure and process interaction and which describes interprocess interaction by message transmission and parallel activity as the arbitrary interleaving of indivisible events.

Two major capabilities are required of a formal modelling scheme for parallel systems with dynamic structure. First, such a scheme must offer a means for describing the components which might exist in the system at some time or, in DPMS terminology, the processes which might be instantiated in the system at some point during the

course of its operation. Descriptions of potential processes must specify both their (potential) behavior and their possible points of connection to the communication pathways by which they may interact with other processes in the system. The modelling scheme must also provide for the description of the system's configuration at a given time and for describing changes in that configuration. A configuration description should include a representation of the currently active components, the current intercomponent communication pathway configuration and the current states of the individual components and of intercomponent communication. In the terminology of the Dynamic Process Modelling Scheme, a configuration description includes a representation of the currently instantiated processes, the current interprocess channel configuration and the current states of the individual processes and links.

The first of these requirements is fulfilled in the Dynamic Process Modelling Scheme by process templates which are used to describe the behavior and interaction possibilities for potential processes in a DPMS model. Each process template represents a class of potential processes; at any given time the modelled system may include zero or more instantiations of any particular process class. A process template is specified in a simple abstract programming language called the Dynamic Modelling Language or DYMOL. DYMOL, discussed in full detail in Chapter III,

contains a set of control constructs, including conditional branching instructions which allow for testing the current system configuration, the last message received by the process or the result of some computation internal to the process. The latter is essentially a nondeterministic branch since internal process computation is not explicitly represented in the modelling scheme. DYMOL also includes a set of instructions for process creation and destruction, alteration of interprocess channel connectivity and interprocess message transmission, which additionally serve to define the potential points of connection between a process and the interprocess communication channels. All the DYMOL instructions have precisely defined formal semantics related to their effect upon the configuration of a modelled system.

The second requirement for a modelling scheme for parallel systems with dynamic structure is fulfilled in DPMS by the instantaneous configuration concept, presented in detail in Chapter IV. An instantaneous configuration represents the complete state of a modelled system at some point in time, largely in terms of the current configuration matrix, a dynamically-structured matrix whose indices represent the currently active (instantiated) processes in the system and whose entries indicate the current pattern of interprocess communication connectivity. The instantaneous configuration also includes information regarding the

current state of each active process and each link in the system. The formal semantics of DYMOL specify the permissible sequences of instantaneous configurations, or computations, which may occur for a given collection of process templates and a given initial instantaneous configuration. Various DYMOL instructions may add or delete rows and columns of the configuration matrix or may alter its entries. A given DPMS model, consisting of a set of DYMOL process templates and an initial instantaneous configuration, is completely characterized by its computation set, the set of all possible sequences of instantaneous configurations which it may produce.

Process template definitional facilities and instantaneous configuration descriptions supply the Dynamic Process Modelling Scheme with the capabilities required to formally model parallel systems with dynamic structure. In succeeding chapters we discuss these capabilities in complete detail, then explore the use of DPMS in investigating parallel systems with dynamic structure.

CHAPTER III

PROCESS TEMPLATES AND THE DYNAMIC MODELLING LANGUAGE

Process templates are employed in the Dynamic Process Modelling Scheme to describe the various types of processes which might be instantiated during the operation of a parallel system with dynamic structure. These templates specify the behavior and interaction possibilities for the potential processes of a DPMS model. Process templates are composed using the Dynamic Modelling Language, or DYMOL, an abstract programming language. In this chapter we discuss process templates and present the Dynamic Modelling Language. Syntax and informal semantics for the instructions and control constructs of DYMOL are given here; the formal semantics for DYMOL appear in Chapter IV.

Process Templates and DPMS Processes

Each process template in a DPMS model describes a class of potential processes; at any given time the modelled system may contain zero or more instantiations of processes from any particular process class. All processes of a given

class are described by the same DYMOL program and hence have exactly the same form and potential behavior. Each instantiated process, however, has a unique process identifier and thus may be distinguished from other processes, whether or not they are of the same process class.

Processes, as conceptualized in DPMS and described by their DYMOL programs, are composed from a small set of basic structures and operations. Before turning to the Dynamic Modelling Language itself, which specifies the set of operations possible for a DPMS process, we will describe the basic structures common to all process templates in the Dynamic Process Modelling Scheme.

A DPMS process may have any or all of three different types of storage structures. One of these is the standard simple variable, a named storage location which either may be empty or may contain a single value from some domain of possible values. For processes in a DPMS model, this domain consists of the (infinite) set of possible process identifiers for the model. A second storage structure found in DPMS processes is the set variable, a named storage location which either may be empty or may contain any number of values from the domain of possible values described above. The values in a DPMS set variable are unordered, the same value may be contained multiple times within a given

set variable, and there can be an unbounded number of values in any such variable. Thus, the set variable is actually a multiset [Knut69] or bag [Cerf72]. The third storage structure found in a DPMS process is the buffer[1], a single distinguished storage location which has a special role in the message transmission activities of the process. The buffer is similar to the simple variable in that it either may be empty or may contain a single value, although the domain of values which the buffer may contain is the set of message types defined for the modelled system. However, in addition to participating in DYMOL buffer assignment statements, the buffer is also the implied source and sink for the message transmission activities of the process. That is, whenever a process sends a message, the message which it transmits is exactly the current contents of its buffer and whenever a process receives a message, the reception is accomplished by the placement of the message into the process' buffer. The possible manipulations of the contents of simple variables, set variables and buffers within a DPMS process are discussed below in the descriptions of the DYMOL instructions which effect those manipulations.

The port is the final type of basic structure found in DPMS processes. Ports, like buffers, are used in process

[1]The buffer concept used in DPMS is essentially that originated by Riddle [Ridd76].

message transmission activities. Each port in a DPMS process represents a potential point of connection between the process and the interprocess communication channels of the modelled system. A process may send messages into these channels through certain of its ports, called outbound ports, and receive messages from the channels through its other ports, called inbound ports. The DYMOL message transmission instructions take ports as their operands. This means that a given process in a DPMS model need not know the identity of the process or processes to whom it is sending messages or from whom it is receiving them. Rather, a message sent through one of its outbound ports may be received by any process which is connected to that port via an interprocess communication channel. Similarly, a message received through one of its inbound ports could have been sent by any process connected via some channel to that inbound port.

This reliance upon ports in describing message transmission activity has two important consequences for the Dynamic Process Modelling Scheme. First, it makes possible the definition of process classes, since message transmission is not specific to the identities of senders or recipients. This capability, in turn, is essential to the fully general representation of parallel systems with dynamic structure, since without it any representation would be limited to only those situations whose complete set of

processes and their interaction structure could be entirely foreseen. Such a limitation would prevent the scheme's use in studying many of the most interesting questions related to parallel systems with dynamic structure, specifically those regarding the possible configurations which such a system might exhibit during the course of its operation. Secondly, the DPMS port concept permits the separate representation of message transmission and message routing activity in models of parallel systems with dynamic structure. Certain DYMOL commands may be used to connect or disconnect pairs of ports, i.e., establish or close interprocess communication channels. The resulting interprocess connectivity structure determines the possible recipients and originators, respectively, for messages sent and received by a process via its execution of certain other DYMOL commands. This capability, too, is vital for dynamic process modelling, since it permits external determination of a process' communication partners. If only the process itself were able to determine the set of other processes with which it could communicate, the range of parallel systems with dynamic structure which could be represented would be greatly diminished and questions such as those regarding protection and communication pathway integrity could not be addressed. Of course, appropriate utilization of the DYMOL instructions for modifying interprocess communication channel connectivity permits the DPMS representation of processes which do completely control the

source and destination of their message transmissions. However, the port concept allows the Dynamic Process Modelling Scheme to be used in studying a much broader range of parallel systems with dynamic structure as well.

A final aspect of ports concerns their fundamental relationship to links, the entities discussed in the preceding chapter which mediate message transmission activity among the asynchronous processes within a DPMS model. A unique link is associated with each outbound port of each process instantiated in a DPMS model and is the implicit initial destination of every message sent through that port. Messages reside in the link until such time as some process requests the receipt of a message through an inbound port which is connected to the link; the channel connecting an outbound port to an inbound port is actually a connection between the associated link and the specified inbound port. If such a request has already been lodged prior to the arrival of a message in the (previously empty) link, that message is immediately transferred to the requesting process. On the other hand, several messages may be stored in the link as a result of multiple message transmissions through the associated outbound port. There they await forwarding to a receiving process which requests a message at some future time. The association of links and outbound ports described here is not complete, however. The deletion of a process from a DPMS-modelled system naturally

results in the disappearance of all of that process' ports, but does not necessarily bring about the dissolution of the links associated with the outbound ports of the process. This feature of the Dynamic Process Modelling Scheme permits messages sent by a process to later be received by other processes even though the sender has, in the interim, been deleted from the system. Thus, it serves to further expand the range of parallel systems with dynamic structure representable in DPMS. Of course, appropriate use of the DYMOL instructions for modifying interprocess communication channel connectivity permits the representation of instances where messages are rendered unreceivable by the destruction of their sender; the conditions governing the dissociation of ports and links are described in detail in the discussion of the DYMOL DESTROY instruction which appears later in this chapter.

DYMOL Basics

A Dynamic Process Modelling Scheme process template is specified by a program written in the Dynamic Modelling Language and labelled with the process class identifier naming the process class represented by the template. In the Algol-based syntax of DYMOL, a program consists of a statement followed by a period. Thus, a DPMS process template is described by the following Backus-Naur Form

(BNF) production:[1]

<process template> ::= <process class id> : <statement> .

The statement may be any single DYMOL statement or a block, i.e., a list of DYMOL statements separated by semicolons and bracketed by the reserved words 'BEGIN' and 'END'. A block may appear anywhere that a statement may legally appear in a DYMOL program.

The process class identifier which labels a DYMOL process template is an alphanumeric string beginning and ending with an alphabetic character[2]. Other DYMOL identifiers, such as those for ports, message classes, variables and statement labels, are alphanumeric strings beginning with an alphabetic character. No DYMOL identifier may be identical to a Dynamic Modelling Language reserved word, i.e., those words used to denote instructions or control constructs in the language. In the interest of brevity and simplicity, DYMOL, unlike Algol, does not require declaration of identifiers used within a process template definition. While such declarations would be

[1]The complete BNF definition of DYMOL, parts of which appear throughout the remainder of this chapter, may be found in Appendix A. Both Backus-Naur Form and Algol are described in [Naur60].

[2]Alphabetic characters in DYMOL include the lower case Roman letters and the symbols '_' and '#'. Process class identifiers are required to end with an alphabetic character in order to accomodate the formation rule for process identifiers, discussed along with the CREATE instruction below.

desirable in an implementation of DYMOL, they are deemed unnecessary to our purpose of succinctly describing formal PSDS models.

Although every DYMOL statement in a DPMS process template, other than a block, is prefixed by a statement label, these labels are not used in specifying the control flow of a DYMOL program; DYMOL has no go-to statement. Instead, the labels are used in representations of DPMS process states (as discussed in the next chapter) and DYMOL control flow is specified using a set of Algol-like structured control constructs. These include iteration, selection and conditional constructs, all of which resemble standard programming language constructs but have been adapted to the task of PSDS modelling.

DYMOL provides two types of iteration, infinite and indefinite. Infinite iteration is a modelling concept useful for describing the behavior of a process which continually repeats the same (arbitrarily lengthy or complex) behavior. While conceptually such a process is non-terminating, in reality its behavior will only be repeated some number of times until an external agent causes it to cease operation -- as in the case of an operating system halted by system shutdown or a PSDS process which is destroyed by some other process. Thus, we are using the term 'infinite' here in an informal, suggestive sense rather

than with its formal, mathematical connotation. Similarly, in the infinite iteration construct, whose BNF description is:

<infinite iteration statement> ::= DO FOREVER <statement>

the DYMOL reserved word 'FOREVER' is used figuratively rather that literally.

The DYMOL construct for indefinite iteration has the reasonably standard BNF specification:

<indefinite iteration statement> ::=

WHILE <condition> DO <statement>

Its semantics are those typical in Algol-like languages; the given statement is repeated as long as the specified condition holds at the beginning of each repetition. The possible conditions for controlling the iteration are, however, peculiar to the modelling of parallel systems with dynamic structure. One possibility for the iteration condition is the DYMOL reserved phrase 'INTERNAL TEST', which indicates that the iteration is nondeterministic, continuing for some random number of repetitions. This is one instance of DYMOL's use of nondeterminism to represent the possible outcomes of the unelaborated, internal computation of processes in a DPMS model and reflects the abstraction inherent in the DPMS focus upon interaction issues in parallel systems with dynamic structure. A second possible iteration condition takes the form:

BUFFER = <message class>

and causes the iteration to continue as long as the process buffer contains a message of the specified class. The third possibility is for a DYMOL indefinite iteration to depend upon the current configuration of the PSDS being modelled, in particular upon the existence of a specific process in the system. This possibility is encoded in DYMOL in the form:

<process id expression> IN A

where <process id expression> is either a process identifier (i.e., the unique name of a single instantiation of some process class) or a simple variable containing a process identifier. The DYMOL reserved word 'A' refers to the set of all currently active, i.e., instantiated, processes in a DPMS model, a concept which is discussed more fully in the next chapter.

One of the selection constructs available in DYMOL can also be considered to be an iteration construct, providing a means of performing definite iteration in a DYMOL program. This, the total selection construct, is phrased according to the BNF production:

<total selection statement> ::=

FOR ALL <value selection> DO <statement>

where <value selection> is an assignment operation as discussed below. The FOR ALL construct causes the statement embedded within it to be repeated once for each value represented by the entity to the right of the assignment

operator in the value selection. That entity may either be a variable (typically a set variable since no iteration would occur if it were a simple variable) or an expression representing some set of values. Prior to each repetition, one previously unselected value is chosen from those represented by the entity and assigned to the (simple or set) variable appearing to the left of the assignment operator in the value selection. The iteration halts when each value represented by the entity has been selected one time and assigned to the variable. In particular, if the entity is initially empty, the embedded statement is never executed, whereas if the entity initially contains <x,x,y> the embedded statement is executed three times.

Two of the three distinct types of assignment defined in DYMOL may appear as the value selection part of a total selection construct. These are described by the BNF productions:

<assignment> ::= <var id> := <expression> |
<var id> :- <var id>

in which <var id> may be an identifier for either a simple variable or a set variable[1]. The first form of the assignment operation is similar to that normally found in

[1]The identifiers for set variables are distinguished syntactically from other DYMOL identifiers by having '/' as their first and last characters. Thus, 'var' could be a simple variable identifier while '/var/' would be an identifier for a set variable.

programming languages, causing a copy of one of the values represented by the entity appearing to the right of the ':=' operator to be placed into the variable to its left. The expressions which can appear to the right of the ':=' may have simple or set variables, sets of values (bracketed by '{' and '}') or the set 'A' of all currently instantiated processes as terms. The terms may appear singly or be combined using the infix binary operators for union (denoted by '+') and set difference (denoted by '-'). The syntax for these expressions is detailed in Appendix A. This assignment operation causes replacement of the previous value in the case of a simple variable and augmentation of the previous set of values in the case of a set variable. The second form of the DYMOL assignment operation is a destructive assignment, causing a value to be removed from the variable appearing to the right of the ':-' operator and placed into the variable to its left. The third type of DYMOL assignment is for the special case of altering the contents of the process' buffer. This is specified according to the BNF production:

<buffer assignment statement> ::=

SET BUFFER := <message class>

where the operand is one of the finite number of message types defined for a given DPMS model. The ':=' operator has the same semantics in this case as it does in the general non-destructive DYMOL assignment operation.

The second form of the DYMOL selection construct is non-iterative but otherwise essentially identical in its behavior to the total selection construct. Called the partial selection construct, its form is given by the BNF production:

<partial selection statement> ::=

FOR SOME <value selection> DO <statement>

where <value selection> can be an assignment of any of the forms legal in the total selection construct. The embedded statement is executed once, following successful completion of the assignment specified by the value selection. As in the case of the total selection construct, partial selection results in a null operation if the entity appearing to the right of the assignment operator in the value selection part is initially empty.

The final type of DYMOL control construct is the conditional, which has the two traditional forms specified by the BNF productions:

<conditional statement> ::=

IF <condition> THEN <simple statement> ELSE <statement> |

IF <condition> THEN <statement>

The <condition>, which determines what action occurs as a result of the execution of a conditional, may be of any of the varieties legal for an indefinite iteration condition. Thus, the conditional may be nondeterministic ('INTERNAL TEST') or may be dependent upon current buffer contents

('BUFFER = ...') or current system configuration ('... IN A'). The requirement of a <simple statement> in the THEN clause of the first form of the conditional is the traditional Algol solution to the ambiguity problem resulting from conditionals embedded within conditionals. It represents no real limitation since any statement or list of statements becomes a DYMOL <simple statement> when bracketed by the reserved words 'BEGIN' and 'END'.

These basic DYMOL features, including variables, assignments and control constructs, exist only for the purpose of supporting the six distinguished DYMOL instructions which describe process interaction. We now consider those instructions, grouped in pairs according to the particular aspect of process interaction to which they relate.

DYMOL Process Structure Instructions

In order to model parallel systems with dynamic structure, some means for adding processes to, and deleting processes from, the modelled system must be available. In the Dynamic Process Modelling Scheme, these alterations to the process structure of a PSDS are accomplished through the use of the DYMOL instructions CREATE and DESTROY.

The DYMOL instruction by which a process can cause a

new process of a specified class to be instantiated in a DPMS model has the form given by the BNF production:

<create statement> ::=

CREATE <process class id> <process ref var>

where <process ref var> is a simple variable and <process class id> is the identifier attached to some process template in the DPMS model. Execution of a CREATE instruction by a process in a DPMS model causes a new process of the class specified by the <process class id> to be created and assigned a unique process identifier. As part of this creation operation, each outbound port of the new process is connected to a distinct, newly created, empty link. The newly assigned process identifier is then placed into the <process ref var> specified in the CREATE instruction and the newly created process is added to the modelled system, where it is immediately eligible to begin operation by executing the first instruction of its DYMOL program. This immediate initiation of the new process is a DPMS convention intended to provide maximal modelling generality. Situations in which the initiation of process execution is a separate operation from instantiation, can be modelled in DPMS with appropriate use of the DYMOL channel structure and message transmission instructions, as indicated in the example of Figure III-2, below.

The process identifier assigned to an instantiated DPMS process consists of its process class identifier suffixed

with a positive integer. Hence, arbitrarily many unique process identifiers can be generated and every instantiated process can be assigned a unique process identifier. In the Dynamic Process Modelling Scheme, no assumption may be made about the process identifiers generated by executions of the CREATE command aside from the uniqueness of those identifiers. In particular, no ordering property applies to the generation of process identifiers, hence it cannot be assumed that any specific process identifier will eventually appear as the result of a series of CREATE instruction executions. Nor do the integer suffixes of the process identifiers for processes instantiated in a DPMS model at a given point in time provide any information regarding the number or sequence of CREATE command executions which have occurred in the modelled system. The DPMS modeller does have a very limited amount of control over the process identifiers assigned to certain processes in a DPMS model, as will be seen in the next chapter.

A process in a DPMS model can cause the deletion of any process from the modelled system by executing a DYMOL instruction of the form specified by the BNF production:

<destroy statement> ::= DESTROY <process id expression>

where the instruction's operand is any DYMOL entity which will evaluate to a process identifier. Such a process identifier expression could be an actual process identifier, or a simple variable, or the DYMOL reserved word 'ME'. The

latter possibility allows a process to cause its own deletion from the modelled system without requiring that it possess its own process identifier, and is necessary since, in general, a process has no knowledge as to what process identifier was assigned to it at the time of its instantiation.

Execution of a DESTROY instruction causes the specific process designated by the instruction's operand to be removed from the modelled system. It also brings about the dissolution of any interprocess communication channels connected to the inbound ports of the designated process. In addition, any empty links attached to the outbound ports of the designated process may be deleted from the system, along with any interprocess communication channels emanating from them. Non-empty links attached to outbound ports of the designated process are not deleted, although the ports themselves, being part of the process, do disappear.[1] Thus, although no new messages can be introduced into these links (since their message source has been destroyed), the messages which they already contain remain available for future reception by any process whose inbound ports are, or become, connected to them. Such surviving links may be deleted from the system if and when they eventually become

[1]The link removal rule given here differs slightly from that implied by the formal semantics described in Chapter IV. The behaviors possible under the two versions are, however, identical.

empty, though removal of empty links is only a cosmetic issue, having no impact on the behavior of the modelled system.

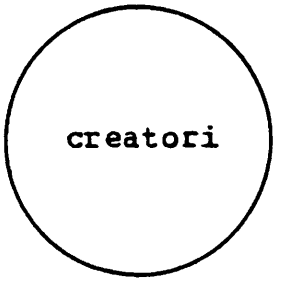
The DESTROY instruction has as its operand an expression which can evaluate to a process identifier, thus designating a specific process to be deleted from the modelled system. However, in all forms other than the reserved word 'ME', this process identifier expression may evaluate to something which does not designate a process instantiated in the system at the time that the DESTROY instruction is executed. This can occur, for example, if the specified process has already been deleted from the system or if the simple variable indicated as the process identifier expression is empty. In such cases, the instruction produces a null operation. Thus, unlike the CREATE instruction which always alters the process structure of a DPMS model when executed, a DESTROY instruction execution may in some instances have no effect upon the modelled system.

A Process Template Example

Having finally described enough of DYMOL to allow for the construction of a reasonably interesting process template, we may now consider the example template of Figure III-1. This template describes the class of potential processes whose process class identifier is 'creator', represented formally by the DYMOL program and informally by the graphical depiction beneath it. Each process of this class, when instantiated, will proceed to instantiate some number of processes of class 'task', then subsequently delete from the system all the task processes which it had created and, finally, delete itself from the system. The number of task processes created and subsequently destroyed by each creator process is determined based upon some computation internal to that creator process and, thus, this behavior is modelled using the nondeterministic version of the indefinite iteration construct. Each creator process remembers what task processes it has created by using the set variable '/tasks/', into which it stores the process identifiers generated by successive executions of the CREATE instruction[1]. This information is then employed in the total selection construct of statement c4 to assure the

[1]The use of the non-destructive assignment operator in statement c3 and of the destructive assignment operator in statement c4 are merely illustrative here. Either or both of these choices could be reversed without affecting the behavior of the creator processes.

```
creator: BEGIN
     c1:   WHILE INTERNAL TEST DO
             BEGIN
     c2:       CREATE task new_name;
     c3:       /tasks/ := new_name
             END;
     c4:   FOR ALL victim :- /tasks/ DO
     c5:     DESTROY victim;
     c6:   DESTROY ME
         END.
```



Example Process Template for 'creator'

Fig. III-1

destruction of each and every process previously created. The DPMS abstraction of internal process computation is clearly illustrated by this example; although the creator processes could well perform many other functions and evidently do at least compute the number of task processes they wish to create, only those aspects of their behavior which impact process interaction are explicitly represented in the DYMOL process template.

DYMOL Channel Structure Instructions

The Dynamic Modelling Language provides two instructions, ESTABLISH and CLOSE, for altering the channel structure in a DPMS model of a parallel system with dynamic structure. These instructions furnish the Dynamic Process Modelling Scheme with the capability to represent the dynamic aspects of interprocess communication connectivity in the operation of a PSDS.

The DYMOL instructions for opening and closing interprocess communication channels have the form indicated by the BNF productions:

<establish statement> ::= ESTABLISH <port name> <port name>

and:

<close statement> ::= CLOSE <port name> <port name>

respectively. The port names which are these instructions' operands consist of port identifiers prefixed by process

identifier expressions as indicated by the BNF production: <port name> ::= <process identifier expression> . <port id> Thus the port name operand in an ESTABLISH or CLOSE instruction refers to a particular port of a specific process, which may be the process executing the instruction (e.g., when the process identifier expression is 'ME') or any other process in the modelled system. This provides DPMS processes with the capability to alter the communication channel connectivity of any pair of processes, which in turn allows DPMS to represent a wide variety of situations with respect to the control of interprocess communication pathways. Depending upon the way in which the DYMOL channel structure instructions are employed, processes of a given DPMS process class may have complete control, partial control or no control over the originators of messages which they receive or the recipients of those which they send. This range of possibilities is illustrated in the example of Figure III-2, which is discussed in the next section.

Execution of an ESTABLISH command causes an interprocess communication channel to be created, connecting the link associated with the outbound port which is the instruction's first operand to the inbound port which is its second operand. Since DPMS process identifiers are unique, the port names unambiguously identify the pair of ports to be joined and the port name of its associated outbound port

unambiguously identifies the link participating in the connection. Messages already in the link at the time that the ESTABLISH instruction is executed, as well as any messages subsequently sent through its associated outbound port, are all eligible to be received through the communication channel created as a result of the ESTABLISH instruction's execution. This convention extends to the case of a link containing messages sent by a process which no longer exists in the modelled system due to the intervening execution of a DESTROY instruction. Since the uniqueness of DPMS process identifiers allows the unambiguous identification of a link by the port name of its associated outbound port, an ESTABLISH command specifying that port name as its first operand can effect a connection with the link despite the fact that the process, and hence the outbound port, with which it had been associated no longer exists. Thus, even messages sent by a no longer extant process, which are already in a link at the time that a channel is connected to that link, are eligible to be received through the newly established channel. However, if either the link specified by its first operand or the inbound port specified by its second operand does not exist, or if a channel already connects the link and inbound port which its operands specify, execution of an ESTABLISH instruction results in a null operation. As this convention suggests, at most one channel may connect a pair of process ports in a DPMS model, multiple channels between ports

providing no advantage due to DPMS message transmission semantics. The convention also reflects the fact that the alternative possibilities of deleting empty links whose associated processes have been destroyed and of retaining such links and perhaps eventually creating communication channels to them result in formally identical behavior for DPMS models. This fact is also due to the semantics of the DYMOL message transmission instructions, which are described in the next section.

Execution of the DYMOL CLOSE instruction has an effect complementary to that of the ESTABLISH instruction. It causes the dissolution of the communication channel connecting the link associated with the outbound port which is the instruction's first operand and the inbound port which is its second operand. Thus messages sent through the specified outbound port can no longer be received through the specified inbound port. Such messages can, however, be stored in the link and later received through the same inbound port should it once again become connected to the link due to some later ESTABLISH instruction execution. A CLOSE instruction execution results in a null operation if either the link or the inbound port specified by its operands does not exist or if no channel connects the designated link and inbound port.

DYMOL Message Transmission Instructions

The final two DYMOL instructions are those used in describing the actual interprocess communication and coordination in a parallel system with dynamic structure. These instructions, SEND and RECEIVE, have been adapted from similar instructions in Riddle's Program Process Modelling Language [Ridd76] to represent interprocess message transmission in DPMS.

The DYMOL instruction by which a process may send a message, making it available for reception via some interprocess communication channel, has the BNF specification:

<send statement> ::= SEND <port id>

where the instruction's operand is the identifier for some outbound port of the process executing the instruction. Since DYMOL has no declaration statements, the appearance of an identifier as the operand of a SEND instruction serves to indicate that it is a port identifier for an outbound port. Observe that, because its operand is a port identifier, a SEND instruction cannot directly specify the recipients for the message which is to be sent. Such message routing is controlled by the DYMOL channel structure instructions and, thus, as noted in the previous section, may be determined totally, partially or not at all by the sending process.

Execution of a SEND instruction by a process causes the current contents of the process' buffer to be sent through one of the process' outbound ports as specified by the instruction operand. A SEND instruction execution leaves the buffer's contents unchanged, i.e., SEND is non-destructive. A copy of the message will thus, in effect, be transferred from the process' buffer to the link associated with the specified outbound port. There the message may await eventual transmission through an interprocess communication channel to some recipient process. Execution of a SEND instruction always immediately results in this movement of a message from the process' buffer to the link associated with the specified outbound port, never necessitating the delay or suspension of the process executing the instruction. Only in the case of an empty process buffer does a SEND instruction execution result in a null operation.

The DYMOL instruction used by a DPMS process in order to request receipt of a message has the form indicated by the BNF production:

<receive statement> ::= RECEIVE <port id>

where the instruction's operand is the identifier for some inbound port of the process executing the instruction. Here again, since DYMOL has no declaration statements, the appearance of an identifier as the operand of a RECEIVE instruction in fact declares it as a port identifier for an

inbound port of the process. Also, as in the case of SEND, the use of an inbound port as its operand gives the RECEIVE instruction no direct control over the originators of the message it seeks to receive. Rather, the DYMOL channel structure instructions provide an independent mechanism for controlling this phase of message routing, which mechanism may or may not be wielded by the receiving process.

Execution of a RECEIVE instruction by a process serves to lodge a request for reception of a message on behalf of the process executing the instruction. Fulfillment of this request will result in a message being placed into the process' buffer. Until such time as the request has been fulfilled, however, the process executing the RECEIVE instruction is "suspended", i.e., not allowed to execute any further instructions. Thus, execution of the RECEIVE instruction may have various effects, depending upon conditions within the modelled system at the time of, and subsequent to, that execution. In particular, if message availability and interprocess connectivity in the modelled system are never such that the request can be fulfilled, the process executing the RECEIVE instruction will be indefinitely suspended, i.e., effectively halted, and never allowed to execute any further instructions.

The conditions necessary to the fulfillment of the request represented by a RECEIVE instruction execution

involve the status of links and channels in the modelled system. In executing a RECEIVE instruction, a process designates an inbound port through which it seeks to receive a message, namely that port specified as the instruction's operand. The request for a message which is lodged as a result of that instruction execution can be fulfilled whenever at least one link containing one or more messages is connected to the designated inbound port by an interprocess communication channel. Thus, if some link connected to the designated inbound port contains a message at the time that the RECEIVE instruction is executed, the request can be fulfilled immediately and the requesting process need not be suspended. Otherwise, the requesting process will be suspended until either a message is sent into one of the links connected to the designated inbound port or a new interprocess communication channel is established connecting the designated inbound port to some link containing one or more messages. Since both of these possibilities depend upon the activities of other processes in the modelled system, the RECEIVE instruction provides a DYNOL mechanism for coordination of processes in a DPMS-modelled parallel system with dynamic structure.

At such time as the necessary conditions for the fulfillment of a process' message request are satisfied, the actual delivery of a message to that process proceeds in the following manner. First, one link is nondeterministically

selected from among those which contain one or more messages and are connected to the designated inbound port by interprocess communication channels. Then one message is chosen, again nondeterministically, from those residing in the selected link. This message is removed from the link and placed into the buffer of the requesting process. This completes the execution of the RECEIVE instruction executed by the the process, which is then allowed to proceed to the execution of the next instruction in its DYMOL program.

It should be observed that, although it may bring about the indefinite suspension of a process, execution of a RECEIVE instruction can never result in a null operation. Completion of the RECEIVE instruction execution always causes delivery of a message, thereby altering the contents of some link and storing some message into a process' buffer. This latter aspect of the instruction operation may or may not alter the buffer contents, depending upon the previous contents of that buffer.

Consideration of the message transmission procedure from the viewpoint of the links may serve to further clarify the operation of the DYMOL message transmission instructions. Each link in a DPMS model is continuously prepared to accept a message resulting from a SEND instruction whose operand designates the outbound port with which the link is associated. Such messages are added to

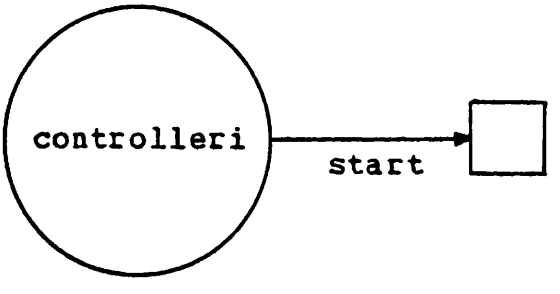
its unordered collection (i.e., bag or multiset) of messages by the link. At the same time, each link, as long as it contains at least one message, is perpetually scanning the set of inbound ports to which it is connected by interprocess communication channels, watching for message requests. When the execution of a RECEIVE instruction causes a message request to appear for the inbound port designated by the instruction's operand, the first link to notice the request will fulfill it by removing a message from its collection and transmitting it through the appropriate channel, thereby placing it in the buffer of the requesting process. Of course, exactly when a given request is noticed by some link depends upon which links are scanning (i.e contain at least one message) and which links can observe the designated inbound port (i.e., are connected to it by a channel). Nondeterminism in the selection of the link and message fulfilling a message request arises from the assumption that links scan inbound ports in random order, with no predictable timing, and that a nondeterministic choice is made from among a group of links which notice a given message request simultaneously.

An Illustrative Example

Having now completed the presentation of process templates and DYMOL, we briefly consider the example templates of Figure III-2. These templates illustrate a variety of DYMOL constructs and features as well as some specific points which have arisen in the preceding sections.

The process template of Figure III-2a describes the process class 'controller'. This class is quite similar to the 'creator' class defined by the template of Figure III-1. However, processes of class 'controller' each have an outbound port with port identifier 'start', as indicated by the DYMOL SEND instruction (c7) and the outward-directed arrow of the accompanying graphical representation. (The box at the end of that arrow represents the link associated with the outbound port.) This port is employed by controller processes to send a message of type 'go' to each of the processes which it has created (in instructions c4 through c7). Of course, the process can actually only send these messages through its 'start' port into the link associated with that port. The fact that these messages are eventually received by the created processes depends upon the behavior of those processes (discussed below) and the set of interprocess communication channels connected to the link associated with the 'start' port. Since the controller is the only process in Figure III-2 which establishes and

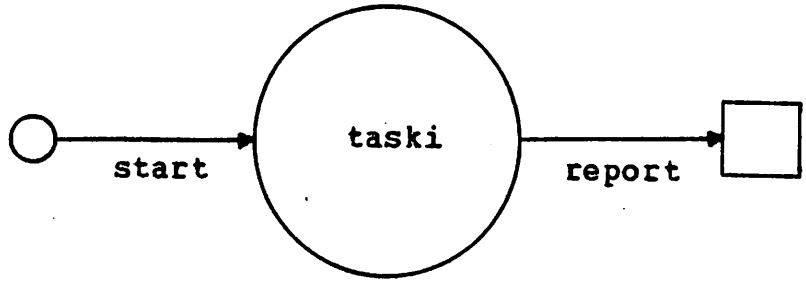
```
controller: BEGIN
       c1:    WHILE INTERNAL TEST DO
                BEGIN
       c2:        CREATE task new_name;
       c3:        /tasks/ := new_name
                END;
       c4:    FOR ALL t := /tasks/ DO
                BEGIN
       c5:        ESTABLISH ME.start t.start;
       c6:        SET BUFFER := go;
       c7:        SEND start
                END;
       c8:    FOR ALL victim := /tasks/ DO
                BEGIN
       c9:        CLOSE ME.start victim.start;
      c10:        DESTROY victim
                END;
      c11:   DESTROY ME
            END.
```



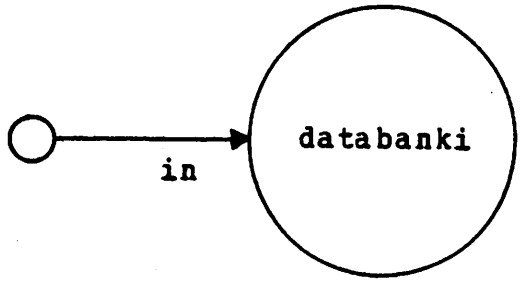
Example Process Template for 'controller'

Fig. III-2a

```
task: BEGIN
  t1:   RECEIVE start;
  t2:   ESTABLISH ME.report databank5.in;
  t3:   WHILE INTERNAL TEST DO
          BEGIN
  t4:       SET BUFFER := data;
  t5:       SEND report
          END
      END.
```




```
databank: d1: DO FOREVER d2: RECEIVE in.
```



Example Process Templates for 'task' and 'databank'

Fig. III-2b

closes channels connected to that link (at c5 and c9, respectively), it is an example of a process which has complete control over the possible recipients of the messages which its sends.

As described in the template of Figure III-2a, a controller process first creates a set of task processes, then connects channels to them and sends them messages, subsequently disconnecting their channels and destroying them, then destroying itself[1]. Alternatively, the controller could have established a channel to each newly created process at the time of its creation (i.e., following statement c2), performing only the sending of messages in the iteration of statement c4, or it could have transmitted one message for each newly created process (by placing statements c6 and c7 immediately after c2) and simply established the appropriate channels in the c4 iteration. Either of these options would have had the same net effect as the one illustrated in Figure III-2a, which is to separate the initiation of the new processes from the event of their creation. Since each process of class 'task' begins with a request for a message which will only be fulfilled when the controller has both sent a message and established a channel to that task process, the newly

[1]Observe that in this example the choice of the non-destructive assignment operator for the iteration of statement c4 is significant, since a destructive assignment in c4 would lead to a null iteration at statement c8.

created tasks will all be suspended until they receive the go messages. Thus, for all practical purposes, the creation and initiation of these task processes are separate events in the system described by the Figure III-2 templates.

As is commonly true in models of parallel systems, no assumptions may be made regarding the relative speeds at which the various DPMS processes in a model of a PSDS complete their successive operations. Thus, it is conceivable that in some instances one or more of the task processes created by a controller will not have received a go message before the controller closes the channel through which that reception was to have been made, and then destroys the task process. Such processes would therefore have been created but never initiated. This possibility remains if the DESTROY instruction (c10) were to be deleted from the 'creator' template. However, this modification would permit those task processes which had been initiated prior to the closing of the channels connecting them to the controller to execute indefinitely, even after the self-destruction of the controller itself. The additional modification of also removing the CLOSE instruction (c9) and, therefore, the now-useless iteration (c8), would allow task processes to even be initiated (i.e., receive a go message) after the self-destruction of the controller. The appropriate choice among these alternatives would depend upon the situation which a modeller sought to represent.

The two templates illustrated in Figure III-2b are relatively simple. Both have inbound ports, declared by their RECEIVE instructions (t1 and d2) and represented graphically by the circle and inward-directed arrow. A process of the 'task' process class, after initiating operation upon completion of the RECEIVE instruction (t1), establishes a channel from itself to a databank process, then sends that process some number of messages of type 'data'. The use of the specific process identifier 'databank5' in the port name operand of the ESTABLISH instruction (t2) indicates the modeller's expectation that this specific process will exist in the modelled system. A limited means for assuring the satisfaction of such expectations is discussed in the next chapter. Since a task process can completely determine the possible recipients for messages which it sends but has no control over the originators of messages which it receives, this process class is one with partial control over its communication partners. If the controller process creating it had established a channel from the task process' report port to some process other than databank5 (as it could by the DYMOL statement 'ESTABLISH new_name.report x.y' inserted after statement c2), then the task process would only have partial control over the recipients of its messages as well.

The template for process class 'databank' is extremely simple. From the DPMS viewpoint, processes of this class

simply receive data messages from task processes. Of course, in any real system being modelled by the templates of Figure III-2 the databank processes would likely perform additional computations using these data messages. However, those internal computations are not explicitly represented in the DYMOL description, which focuses only upon the process interaction aspects of system activity. The databank process described by this final process template is of interest chiefly because, not establishing or closing any channels, it is an example of a process with no control over its communication partners.

Extended DYMOL

In the course of developing the Dynamic Modelling Language, we considered several alternative formulations and a number of additional features for possible inclusion in DYMOL. In response to the potential merits of some of these possibilities, a second version of the language, called Extended DYMOL, was defined. A BNF description of this extended version of the modelling language appears in Appendix B. Although for the remainder of this dissertation we shall restrict our attention to DYMOL as defined in the preceding sections of this chapter, we here briefly discuss the extended version of the Dynamic Modelling Language.

As its name suggests, Extended DYMOL is an upward-

compatible version of the language defined in this chapter. It differs from the basic form of the Dynamic Modelling Language in two significant ways. First, it allows for parameterized process templates, thereby permitting a process to influence the behavior of those processes which it creates. Secondly, Extended DYMOL permits the sending and receiving of process identifiers, port names and other specific information as messages and the use of such messages as operands in Extended DYMOL instructions.

The parameterized template feature of Extended DYMOL necessitates a modification to the form of process templates and a change in the CREATE instruction. Exended DYMOL process templates are allowed to have a parenthesized formal parameter list following the process class identifier in their modelling language definition. The formal parameters, which can either be simple or set variables, can appear as operands in almost any Extended DYMOL instruction other than SEND and RECEIVE, which retain their port identifier operands. The Extended DYMOL CREATE instruction permits the specification of a set of actual parameters to accompany the process class identifier in its operand field. Thus a process may create new processes which, while all of the same class, have somewhat different potential behaviors. In particular, parameterizing can be used to determine what other process or processes a newly created process will be able to interact with, i.e., designate as an operand in an

ESTABLISH, CLOSE or DESTROY instruction. Moreover, parameter values may be passed along through successive process creations so that all the offspring of a given process will share information distinct from that known to the progeny of some other process. The obvious relationship between these properties of parameterized templates and such standard software notions as capabilities [Denn66] and various protection mechanisms was predictably influential in their inclusion in Extended DYMOL.

The expansion of the set of possible messages from a finite number of distinguishable message types to an unbounded number of distinct values, including process identifiers and port names, clearly impacts the amount of information which can be exchanged among processes in a system modelled using Extended DYMOL. The syntax changes required for the implementation of this feature in the modelling language are minimal. Extended DYMOL simply permits the process' message buffer to appear as a term in the expressions of non-destructive assignment statements and allows simple variables to appear on the right hand side of buffer assignments. Here, as in the case of parameterized process templates, the major effect of this modelling language modification is to expand the ability of processes to influence the possible interaction partners of other processes. In particular, a message may now be used by its recipient as an operand in a CREATE, DESTROY, ESTABLISH or

CLOSE instruction. Indeed, this feature of Extended DYMOL serves an information diffusion function for processes already instantiated which is very similar to that afforded the process creation procedure by parameterized templates. Not surprisingly, this expanded message utilization also bears a notable resemblance to capabilities and software protection systems.

The features of Extended DYMOL would certainly expand the descriptive abilities of DPMS, permitting a more natural depiction of certain types of dynamically-structured, complex software systems. However, as we will show in Chapter V, the formal modelling power of DPMS would not be increased through the use of Extended DYMOL. Moreover, the extended version of the modelling language introduces certain technical difficulties in representing configurations of the modelled system. We therefore defer the study of Extended DYMOL to future investigation and in the present work consider only the Dynamic Modelling Language as defined in the previous sections of this chapter and in Appendix A.

CHAPTER IV

THE DYNAMIC PROCESS MODELLING SCHEME

To be useful for studying parallel systems with dynamic structure, a formal modelling scheme must offer both a means for describing the potential components of a PSDS and also a technique for representing the modelled system's configurations and the changes which they undergo during system operation. The Dynamic Modelling Language developed in the preceding chapter provides the Dynamic Process Modelling Scheme with a means for describing potential PSDS components as process class templates. In the present chapter we discuss the DPMS mechanisms for the representation of PSDS configurations and their changes over time, thus completing our presentation of the Dynamic Process Modelling Scheme. This treatment includes the definition of formal semantics for the constructs and statements of DYMOL in terms of their impact upon DPMS configuration representations. The chapter also contains an example illustrating the use of the Dynamic Process Modelling Scheme and indicating its potential as a tool useful in the design and analysis of complex, dynamically-structured software systems.

The Configuration Matrix

The DPMS description of a parallel system with dynamic structure begins with a set of DYMOL process templates defining the potential processes of the system. Having such a set of definitions for the possible components of the PSDS, a representation of the system's configuration, that is, its set of instantiated processes and their interconnection, is still required. Fundamental to the Dynamic Process Modelling Scheme's representation of PSDS configurations is a structure known as the configuration matrix, or C. The configuration matrix is a two-dimensional array whose indices along both dimensions are process identifiers which have been assigned to instantiations of the process classes defined by the templates. The entries in this matrix consist of ordered pairs of port identifiers from processes of the template-defined classes. Both the contents of the matrix entries and the number of rows and columns in C may vary over time, reflecting changes in the configuration of the modelled system.

The indices of the configuration matrix at any point in time form a subset of those process identifiers assigned to processes instantiated in the modelled system over the course of its operation. The column indices of C are precisely the process identifiers for those processes presently instantiated in the modelled system. The indices

of the rows in C consist of these plus the process identifiers of any processes which (1) were instantiated, (2) have since been destroyed and (3) have left the links associated with one or more of their outbound ports still residing in the modelled system. Thus, at any time during system operation, the column indices of C indicate exactly which processes currently exist in the system and hence all the possible message recipients in the present configuration. Similarly, the row indices indentify all those processes whose outbound ports are or were associated with the links currently existing in the system, and hence the originators of all messages which are currently available for reception. Of course, in those instances where all of the links within a modelled system are associated with outbound ports of currently instantiated processes the row and column index sets are identical and C is a square matrix.

The example configuration matrices of Figure IV-1 illustrate these possibilities for row and column indices. Assuming a set of process templates defining process classes 'controller' and 'task' (e.g., the templates of Figure III-2), the first configuration matrix of Figure IV-1 indicates the existence of two instantiated processes in the modelled system, namely controller3 and task5. The absence of any additional rows in this matrix implies that if any other processes have ever been instantiated in the system they

	controller3	task5
controller3	∅	∅
task5	∅	∅

	controller3	task5	task2
controller3	∅	∅	∅
task5	∅	∅	∅
task2	∅	∅	∅

	controller3	task2
controller3	∅	∅
task5	∅	∅
task2	∅	∅

Example Configuration Matrices

Fig. IV-1

have subsequently been destroyed and no non-empty links associated with them remain in the system at the present time. The second configuration matrix, formed from its predecessor by the addition of a row and column, indicates that a new process, task2, has been instantiated in the system. The creation of a process in a DPMS-modelled system always results in the augmentation of C with a row and column, each indexed by the process identifier assigned to the new process. The final configuration matrix of Figure IV-1 could arise from its predecessor due to the destruction of the process task5. As indicated by the column indices of this configuration matrix, only controller3 and task2 would then remain as instantiated processes. However, the presence of the row with index 'task5' in this matrix implies that some link associated with an outbound port of task5 was non-empty at the time of the process' destruction and remains in the modelled system as a potential source of messages.

While the indices of the configuration matrix summarize those aspects of a PSDS configuration related to currently instantiated processes and surviving links, the contents of C describe the current interprocess communication channel configuration of the DPMS-modelled system. Each matrix entry contains a set of ordered pairs of port identifiers. The first port identifier of each pair corresponds to an outbound port of the process whose process identifier

indexes the row in which it appears. The second port identifier of a pair corresponds to an inbound port of the process whose identifier serves to index the column in which it is found. Thus, each ordered pair in a set indicates the existence of an interprocess communication channel which connects the link associated with the specified outbound port and the designated inbound port. When an entry is empty, as was the case for every entry in each configuration matrix of Figure IV-1, no interprocess communication channels connect the outbound ports of the process corresponding to its row with the inbound ports of the process corresponding to its column. It should be noted that configuration matrices are not, in general, symmetric, although symmetry is possible when all entries are empty or when processes with like-named ports, whether or not they are of the same process class, are connected in an appropriate fashion.

The configuration matrices of Figure IV-2 illustrate the encoding of channel information in C[1]. Again assuming process classes 'controller' and 'task', the first of the configuration matrices in Figure IV-2 indicates that the link associated with the outbound port having identifier 'start' of process controller3 is connected to inbound ports having that same identifier in processes task5 and task2.

---

[1]To simplify notation, set brackets around singleton sets in C are omitted throughout this dissertation.

	controller3	task5	task2
controller3	∅	(start,start)	(start,start)
task5	∅	∅	∅
task2	∅	∅	∅

	controller3	task5	task2
controller3	∅	(start,start)	∅
task5	∅	∅	∅
task2	∅	∅	∅

	controller3	task5
controller3	∅	(start,start)
task5	∅	∅

More Example Configuration Matrices

Fig. IV-2

This matrix could result from the execution of two ESTABLISH commands occurring when the configuration matrix was of the form shown in the second example of Figure IV-1. The empty entries in the remainder of the matrix show that no process can send messages to itself and that the two processes of class 'task' cannot send messages to controller3 nor to one another. The second configuration matrix could arise from the first as a result of the closing of the channel connecting the respective start ports of controller3 and task2. In general the closing of a channel removes one ordered pair from the set contained in the corresponding matrix entry; in this instance, that results in the entry becoming the null set, indicating that communication is no longer possible between processes controller3 and task2. The final configuration matrix of Figure IV-2 could have been obtained from its predecessor, or directly from the first matrix, through the destruction of process task2. Since destruction of a process includes the deletion of all its inbound ports, any channels connected thereto are automatically closed by such a destruction. Explicitly closing them before the process is destroyed is not required since process destruction deletes the appropriate column from C whether or not its entries are all empty. Of course, it is also true that the contents of a row in C have no bearing upon its deletion when its corresponding process is destroyed, but the justification in this case differs from the case of the matrix column. The appropriate row in C

will be deleted, at the same time or subsequent to the deletion of the associated process, only if all of the links associated with the process' outbound ports are empty. This condition is independent of the current channel connectivity involving those links. Thus a row with all empty entries might be retained in C due to the presence of non-empty links, while another row could be deleted although it contained one or more non-empty entries because all the associated links, despite their channel connections, contained no further messages and thus could not figure in any future message transmission.

The Instantaneous Configuration

A complete characterization of the current configuration in a parallel system with dynamic structure requires more information than is found in a configuration matrix. While the matrix is sufficient to indicate exactly which processes and links currently exist in the system and also to describe the interprocess communication connectivity among those processes and links, it says nothing as to the current state of the instantiated processes nor the current contents of the links. Therefore, the Dynamic Process Modelling Scheme provides the instantaneous configuration, a structure containing all of this necessary information, as its basic means for representing the configuration of a PSDS at some particular point in time.

In addition to process states, link contents and the configuration matrix, an instantaneous configuration contains a certain amount of redundant information which is notationally convenient for DPMS. The active process set, denoted A, is simply a set consisting of the process identifiers of all currently instantiated processes. These identifiers appear in A in exactly the same order that they occur as column indices in C. Being identical to the column index set of the configuration matrix, A is not formally necessary to the representation of instantaneous configurations. Its convenience in such notations as the DYMOL total selection statement:

FOR ALL t IN A DO ...;

however, warrants its inclusion in DPMS instantaneous configurations.

The state of an instantiated process in a DPMS-modelled system has several components. It is determined in part by the current contents of the process' various memory locations, i.e., its buffer and any variables which are defined by the process template for the class of which it is an instance. In addition to these, the process' state depends upon where it is currently executing in its DYMOL program. This information is conveniently visualized in terms of a location counter, or pointer, indicating the next DYMOL instruction that the process will attempt to execute. In a DPMS instantaneous configuration the state of a process

is formally represented as indicated by the following:

> Definition IV.1: A Dynamic Process Modelling Scheme process state for an instantiated process is denoted by 'q[pi]', where pi is the appropriate process identifier. The process state q[pi] = (lc,v(1),...,v(n),f(1),...,f(n),bu) where:
> lc is the location counter, a statement label from the DYMOL program defined in the process class template corresponding to process pi. This label designates the next DYMOL instruction to be executed by process pi;
> bu is a message type identifier which specifies the current contents of the buffer of process pi;
> v(1),...,v(n) are keyworded multisets specifying the current contents of each of the n (n≥0) variables defined by the DYMOL process template corresponding to process pi; and
> f(1),...,f(n) are a variable number of special purpose multisets, keyworded with *fi* where fi is the label of a total selection statement in the DYMOL program specifying the process template for pi.

Thus, for example, a possible process state for a process task7 of the process class defined by the 'task' process template of Figure III-2b might be denoted as:

q[task7] = (t5,data)

indicating that the process' buffer currently contained a message of type 'data' and that it was about to execute the SEND instruction which is labelled t5 in the DYMOL template for the 'task' process class. Similarly, a process controller4 of the process class defined by the 'controller' process template of Figure III-2a might be in a state designated by[1]:

---

[1]Following [Cerf72] we use the notation <...> in specifying the contents of a multiset, while using ∅ to represent empty multisets as well as empty sets.

q[controller4] = (c5,new_name=<task2>,t=<task7>,victim=Ø,
/tasks/=<task4,task7,task2>,*c4*=Ø,Ø)

where the contents of the various variables which are part of a process of the class 'controller' appear in keyworded form and the final tuple entry indicates that the process buffer is presently empty. The use of the special purpose, keyworded multiset *c4* is described below in the formal semantics for the DYMOL total selection statement.

Using the notion of process state as given by Definition IV.1, the Dynamic Process Modelling Scheme representation for the entire collection of states of the individual instantiated processes in a DPMS-modelled system is defined as follows:

> Definition IV.2: A Dynamic Process Modelling Scheme complete process state, Q, is an ordered n-tuple of process states in which the process states for the n currently instantiated processes are specified in the same order as their corresponding identifiers appear in the active process set, A.

For example, if only the two processes described above, task7 and controller4, were currently instantiated (e.g., A={controller4,task7}), then a complete process state could be denoted by:

Q = (q[controller4],q[task7])

or

Q = ((c5,new_name=<task2>,t=<task7>,victim=Ø,
/tasks/=<task4,task7,task2>,*c4*=Ø,Ø),(t5,data))

presuming the individual process states were identically

those given in the previous example.

In order to specify the contents of the links associated with a given process, it is convenient to define a canonical ordering on the process' links. An obvious ordering for outbound ports, and hence for the associated links, of a process is the order in which they are first referenced (by a SEND instruction) in the DYMOL program corresponding to the process. We call this ordering the natural definitional order for links and may then define the DPMS representation for the current contents of a process' links as follows:

> Definition IV.3: A Dynamic Process Modelling Scheme link state for the links associated with some currently or formerly instantiated process is denoted by 'ls[pi]' where pi is the appropriate process identifier. The link state, ls[pi], is an ordered n-tuple of multisets in which each multiset's contents are the current contents of a link and the multisets appear in the natural definitional order of their corresponding links.

When the link state for a process pi includes many null entries, it is sometimes convenient to represent ls[pi] in a keyworded notation. In this format the various non-empty multisets are listed, keyworded with the port identifiers associated with the links which they represent, while those not listed are presumed to be empty.

Using this definiton of link state, the Dynamic Process Modelling Scheme representation for the current contents of all links in a DPMS-modelled system is then defined as

follows:

> Definition IV.4: A Dynamic Process Modelling Scheme complete link state, L, is an ordered n-tuple of link states in which the link states are specified in the same order that the process identifiers of the processes to which they correspond appear as row indices in the configuration matrix, C.

For example, a possible complete link state in a DPMS-modelled system whose current configuration matrix was the last of those exhibited in Figure IV-1 might be:[1]

$$L = ((\langle go,go\rangle),\emptyset,(\langle data,data,data\rangle))$$

This complete link state would indicate that the link associated with the 'start' port of process controller3 contained two messages of class 'go' while the links associated with the 'report' ports of task5 and task2 contained zero and three messages of class 'data', respectively.

Having developed all its necessary components, we may now define the formal Dynamic Process Modelling Scheme representation for the configuration of a modelled system at a particular instant in time:

> Definition IV.5: A Dynamic Process Modelling Scheme instantaneous configuration, x = (C,A,Q,L) where:
> C is the current configuration matrix,
> A is the current active process set (i.e., the column indices of C),
> Q is the current complete process state, and
> L is the current complete link state.

---

[1]Here and throughout this dissertation parentheses are omitted from tuples consisting of a single empty multiset.

As we have seen in the preceding discussion of its components, the DPMS instantaneous configuration completely characterizes the situation within a modelled parallel system with dynamic structure at any given point in time.

Formal Semantics for the Dynamic Modelling Language

Since the DPMS instantaneous configuration provides a means for describing the configuration of a parallel system with dynamic structure at any single instant, a sequence of these instantaneous configurations could serve to represent the changes which a PSDS can undergo during its operation. Determination of the sequences of instantaneous configurations which could possibly be generated from a specified initial configuration in a DPMS-modelled system depends crucially upon the meanings associated with the DYMOL programs which represent the possible process activity of the system. Thus we now proceed to develop a formal semantics for the DYMOL constructs and statements in terms of their effect upon instantaneous configurations. Having done this we will be able, in the next section, to formalize the notion of computation in DPMS, and to rigorously define the concept of a model in the Dynamic Process Modelling Scheme.

In this presentation, we give the formal semantics for DYMOL in terms of the effect which the execution of each

individual instruction or control construct can have upon the instantaneous configuration of a DPMS-modelled system. Thus we will frequently refer to those parts of an instantaneous configuration which relate to the particular process executing that instruction. We denote this currently executing process by the pseudo-process identifier 'pc', and will accordingly refer to its process state as q[pc], to the location counter within its process state as lc[pc], etc. We will also, in this presentation, make frequent use of the concept of end of statement, which will be denoted 'eos' in the sequel. In particular, in describing the final step in completing a successful instruction execution, we will often write 'lc[pc] <- eos' (read 'location counter is set to eos'), indicating that the currently executing process' location counter is updated in the appropriate fashion for an end of statement condition. This notion serves to abbreviate the several possibilities for the updating of a process' location counter. Each DYMOL control construct contains one or more embedded statements, and the appropriate change to lc[pc] upon completion of those embedded statements varies with the control construct. Iterative constructs, for example, may set lc[pc] to point back to the beginning of the iteration upon completion of their embedded statements, while conditional constructs imply a totally different end of statement behavior. As part of our formal definition for each of the DYMOL control constructs, we indicate the effect of an end of statement

occurring in its embedded statement(s). Such an indication is referred to as an eos directive for the control construct. In those cases where a DYMOL instruction is not the embedded statement of a control construct, i.e., where no eos directive is applicable, eos generally represents the label of the sequentially next DYMOL statement in the process template for pc. If that next statement is a block, and therefore commences with an unlabelled BEGIN, eos simply becomes the label of the first labelled statement within that block. Similarly, when the end of statement condition occurs in the last labelled statement within a block, it implies eos for the statement which that block syntactically represents. Finally, eos for the statement which is a complete DYMOL program, i.e. a statement which is followed by a period in the DYMOL syntax, is simply $\emptyset$, signifying termination of process activity.

The DYMOL assignment instructions have perhaps the simplest semantics of any in the Dynamic Modelling Language. Execution of the buffer assignment statement:

SET BUFFER := msg

causes replacement of the bu[pc] entry of q[pc] with the message class identifier specified in the instruction, then lc[pc] <- eos. Both other forms of the assignment statement begin their execution by evaluating the entity or expression appearing to the right of the assignment operator. This evaluation is straightforward, involving the use in some

cases of values from one or more of the v(i)[pc] in q[pc] and possibly the contents of the active process set, A, found in the instantaneous configuration. One value is then selected from the (possibly singleton or null) set of values resulting from this evaluation and placed into the v(j)[pc] corresponding to the left hand side of the assignment statement. This placement replaces the prior contents of v(j)[pc] unless its keyword is of the form /.../, in which case the placement adds to the contents of the multiset v(j)[pc]. When the initial evaluation results in a null set of values, this placement sets a simple variable's v(j)[pc] to $\emptyset$ while it leaves v(j)[pc] unchanged for a set variable. If the assignment operator is ':-', the selected value is then deleted from the v(j)[pc] multiset corresponding to the right hand side of the assignment. Finally, upon completion of these steps, lc[pc] <- eos.

The DYMOL infinite iteration construct has the form represented by the schema:

ai: DO FOREVER aj: <statement>

Whenever lc[pc]=ai, execution of this instruction simply results in lc[pc] <- aj. Every completion of the statement represented by <statement> in the above schema, i.e., every eos for the embedded statement, results in lc[pc] <- ai. This latter stipulation is an example of an eos directive, in this case the eos directive for infinite iteration.

Indefinite iteration statements in DYMOL have the form suggested by the schema:

ai: WHILE <condition> DO aj: <statement>

Whenever lc[pc]=ai, the condition denoted by <condition> in the above schema is evaluated. This evaluation, which may result in either a 'true' or a 'false' result, is straightforward. It involves either a nondeterministic choice of 'true' or 'false' (if the condition is INTERNAL TEST), consultation of bu[pc] (if the condition is BUFFER=...) or consultation of the active process set A (if the condition is ... IN A). If the evaluation's result is 'true', then lc[pc] <- aj. Otherwise, lc[pc] <- eos. Upon every completion of the embedded statement represented by <statement> in the schema, lc[pc] <- ai. Notice the distinction between this, the eos directive for the indefinite iteration construct, and the previously stated action (lc[pc] <- eos) which is to be taken upon completion of the WHILE statement itself. Clearly, the respective eos's for the control construct statement and the statement embedded within the construct are, in general, completely different.

A simple example may help to clarify the end of statement notion. Consider the following illustrative, if improbable, DYMOL fragment:

```
bi:  DO FOREVER
bj:    WHILE INTERNAL TEST DO
         BEGIN
bk:        a := b;
bn:        /a/ := a
         END
```

The eos for the assignment statement labelled bk here would be bn, since the bk statement is not an embedded statement within a control construct, although it is part of such an embedded statement, and hence no eos directive is applicable. However, eos for bn, the final statement within a block, is eos for the block and therefore we have lc[pc] <- bj, since the block is an embedded statement within an indefinite iteration. Now if evaluation of the INTERNAL TEST yields the result 'false', eos for the indefinite iteration statement, as opposed to eos for the statement embedded within it, is <u>not</u> governed by the eos directive for indefinite iteration. Rather, since the WHILE statement is itself an embedded statement in the infinite iteration construct, eos for it becomes bi, according to the eos directive for infinite iteration.

The DYMOL total selection construct may be schematized as:

ai: FOR ALL <value selection> DO aj: <statement>

Whenever lc[pc]=ai, the process state q[pc] is examined for the existence of an f(i)[pc] whose keyword is *ai*. If no such f(i)[pc] exists, then the right hand side of the assignment represented by <value selection> in the above

schema is evaluated and the result of this evaluation becomes the contents of a multiset which is given the keyword *ai* and added to q[pc]. Completion of this operation is followed by the trivial update to the location counter for pc indicated by lc[pc] <- ai. If the examination of q[pc] reveals that an f(i)[pc] with keyword *ai* exists but is empty, the result is the deletion of *ai* from q[pc] and lc[pc] <- eos. The third possibility, i.e., discovering a non-empty *ai* entry in q[pc], leads to (1) a nondeterministic selection of an item from *ai*, (2) deletion of that item from *ai*, (3) placement of that item into the variable on the left hand side of the assignment operator in the <value selection>, possibly accompanied by its deletion from the entity on the right hand side, all in accordance with the usual semantics for DYMOL assignments and (4) lc[pc] <- aj. The three possible outcomes of the inspection of q[pc] correspond to what may be informally described as the initiation, the completion and the continuation, respectively, of the operation specified by the total selection statement. Upon every completion of the embedded statement represented by <statement> in the schema, the eos directive for the total selection construct specifies lc[pc] <- ai.

The schema for DYMOL partial selection statements is:

ai: FOR SOME <value selection> DO aj: <statement>

Whenever lc[pc]=ai, the assignment specified by the <value

selection> is performed according to the usual semantics for DYMOL assignment statements. If the right hand side of that assignment evaluates to $\emptyset$, lc[pc] <- eos; otherwise lc[pc] <- aj. Upon completion of the embedded statement represented by <statement>, lc[pc] <- eos. It is this eos directive which formally accounts for the fact that the partial selection statement is not iterative.

Two different schemata may be given for the DYMOL conditional statement, namely:

ai: IF <condition> THEN aj: <simple statement>
ELSE ak: <statement>

and:

ai: IF <condition> THEN aj: <statement>

Whenever lc[pc]=ai for either form of the DYMOL conditional statement, the condition represented by <condition> in the schemata is evaluated in exactly the same manner as for the indefinite iteration described previously. For both forms of the statement, if the condition evaluation results in 'true', lc[pc] <-aj. If the result of this condition evaluation is 'false', lc[pc] <- ak in the case of the first schematized form for the statement and lc[pc] <- eos in the case of the second form. The eos directive for all the embedded statements, represented by either <simple statement> or <statement>, in both forms is lc[pc] <- eos.

A typical form for the DYMOL instruction for process

creation is:

ai: CREATE pcid refvar

Whenever lc[pc]=ai a sequence of activities takes place[1]. First a new, unique process identifier, which we will denote by 'pcidx', is generated by appending an integer to the process class identifier, pcid in this case, specified by the instruction's first operand. This newly generated process class identifier is placed into the process reference variable, refvar, just as if the assignment instruction refvar := pcidx had been executed. A new rightmost column and new bottom row are then added to the configuration matrix, C, each indexed with pcidx and containing $\emptyset$ in each of its entries. Next, pcidx is added to the end of the active process set, A, and a new process state, q[pcidx], is added as the new last entry in the complete process state, Q. The elements of q[pcidx] have the values: lc[pcidx] = first statement label in the DYMOL template for pcid, bu[pcidx] = $\emptyset$, and v(i)[pcidx] = $\emptyset$, with one appropriately keyworded v(i)[pcidx] entry corresponding to each variable defined by the pcid template. Following this, a new link state, ls[pcidx] is added as the new last entry in the complete link state, L. This new link state is an n-tuple of $\emptyset$'s, where n is the number of outbound ports defined by the DYMOL template for process class pcid.

---

[1]Since they have no embedded statements, none of the remaining DYMOL instructions has an eos directive as part of its formal semantics.

Finally, lc[pc] <- eos.

A DYMOL instruction for destroying a process in a DPMS-modelled system typically has the form:

ai: DESTROY pi

Whenever lc[pc]=ai, the active process set A is inspected for an occurrence of the process identifier, pi, specified by the instruction's operand[1]. If pi is not found in A, the instruction execution terminates and lc[pc] <- eos. Otherwise, a series of activities takes place. First, pi is deleted from A, the column whose index is pi is deleted from the configuration matrix, C, and q[pi] is deleted from the complete process state, Q. Next the link state ls[pi] in L is inspected, and if all entries in ls[pi] are $\emptyset$, then ls[pi] is deleted from L and the row whose index is pi is deleted from the configuration matrix, C. Finally, unless pi=pc, lc[pc] <- eos.

Typical of the DYMOL commands for opening a communication channel between two ports is the instruction:

ai: ESTABLISH pi.ri pj.rj

Whenever lc[pc]=ai, the configuration matrix, C, is inspected for the existence of an entry, C(pi,pj), corresponding to the two processes specified by the two port

---

[1]In some instances, the operand must be evaluated first, requiring a straightforward inspection of some v(i)[pc] or the use of pc itself if the operand is the DYMOL reserved word 'ME'.

name operands of the instruction[1]. If no such entry is found in C, the instruction execution terminates with lc[pc] <- eos. Otherwise, the ordered pair (ri,rj) is added to the set which is the contents of C(pi,pj). Since this operation is formally a union of the former contents of C(pi,pj) and {(ri,rj)}, the addition of an ordered pair already contained in C(pi,pj) does not alter the contents of the matrix entry. After the new ordered pair has been added to C(pi,pj), lc[pc] <- eos.

A typical form in which the DYMOL complement to the ESTABLISH instruction might appear in a process template is:

ai: CLOSE pi.ri pj.rj

Here too, whenever lc[pc]=ai the configuration matrix, C, is inspected for the existence of C(pi,pj), and if no such entry can be found, the instruction execution terminates with lc[pc] <- eos. If the configuration matrix entry does exist, the ordered pair (ri,rj) is removed from C(pi,pj). Being formally a set difference operation, this removal has no effect upon the contents of C(pi,pj) if (ri,rj) was not an element of the set contained in the matrix entry at the time this CLOSE instruction was executed. Following the removal of the ordered pair from C(pi,pj), lc[pc] <- eos.

The DYMOL message transmission instruction used by a

---

[1]As with DESTROY, evaluation may be required to determine pi and pj from the ESTABLISH instruction's operands.

process for sending a message through one of its outbound ports typically takes the form:

ai: SEND ri

Whenever lc[pc]=ai, the current value of bu[pc] is added to the multiset of ls[pc] corresponding to the outbound port ri. This addition is, in formal terms, a 'bag addition' [Cerf72] and therefore does not alter the contents of the subject multiset if bu[pc]=$\emptyset$. Following this addition to the element of ls[pc], lc[pc] <- eos.

Finally, we describe the formal semantics for the DYMOL message transmission instruction used to request the reception of a message, which typically has the form:

ai: RECEIVE ri

Whenever lc[pc]=ai, a set PL is formed and its contents initially set to be $\emptyset$. Next, the configuration matrix column whose index is pc is scanned, and each entry is examined for ordered pairs with ri as their second entry. For all port names pj.rj such that C(pj,pc) contains an ordered pair (rj,ri) and the link associated with pj.rj is non-empty (i.e., the appropriate entry of ls[pj]$\neq\emptyset$), the port name pj.rj is added to PL. Note that since there can be no duplication of ordered pairs within entries of C, no port name can be added to PL more than one time. If PL is empty at the end of this scan of column pc in C, execution of the RECEIVE instruction terminates with the destruction of PL and lc[pc] <- ai. Thus, when no message is available

to be received, the process is not allowed to advance to the execution of another instruction, i.e., is "suspended". When PL is non-empty at the completion of the scan, however, a sequence of activies ensues. First, an element pk.rk is nondeterministically selected from PL and the set PL is destroyed. Next, one item from the selected link pk.rk, whose contents are specified by the value of the appropriate multiset in ls[pk], is nondeterministically chosen, deleted from the multiset in ls[pk] and made the new value of bu[pc]. Then, if pk is not an element of A and if this step resulted in the link state ls[pk] becoming an n-tuple of β's, the row whose index is pk is deleted from the configuration matrix, C, and ls[pk] is deleted from L. Finally, lc[pc] <- eos.

The DPMS Computation Step

In this chapter we have developed the formal Dynamic Process Modelling Scheme representation for the configurations of parallel systems with dynamic structure and also described formal semantics for the instructions and constructs of the Dynamic Modelling Language in terms of their effect upon that representation. Thus, we are now in a position to give the final DPMS definitions and thereby complete our presentation of the Dynamic Process Modelling Scheme.

The DPMS instantaneous configuration permits the description of the configuration of a parallel system with dynamic structure at any given point during its operation. The changes which such a system may undergo during its operation can therefore be described in terms of sequences of instantaneous configurations. The fundamental unit of such a sequence is the computation step, a pair of instantaneous configurations corresponding to the endpoints of a single basic execution cycle in a DPMS-modelled system.

The basic execution cycle of the Dynamic Process Modelling Scheme proceeds in two steps. At the first step, a process identifier, pi, is nondeterministically selected from the active process set, A, such that lc[pi]≠∅. The second step then consists of performing the operation specified by the statement which is labelled lc[pi] in the DYMOL process template corresponding to process pi. This operation manipulates the instantaneous configuration according to the formal semantics of the DYMOL statement and is considered to be completed at the point that an assignment is made to lc[pi], whether or not that assignment actually alters the contents of lc[pi]. In fact, as examination of the formal semantics of DYMOL will confirm, every basic execution cycle except for those corresponding to the initialization phase of a total selection statement or an attempted RECEIVE statement for a suspended process leads to the alteration of lc[pi].

Using the notion of a DPMS basic execution cycle we may now define the fundamental unit of computation in the Dynamic Process Modelling Scheme as follows:

> Definition IV.6: A Dynamic Process Modelling Scheme computation step, y<i,j> is defined to be x<i>x<j> where x<i> and x<j> are instantaneous configurations, x<i>≠x<j>, and x<j> is obtainable from x<i> through some single DPMS basic execution cycle.

Thus, a computation step serves to represent the minimal change to the configuration of a DPMS-modelled PSDS resulting from some activity on the part of one of its processes. The inequality, x<i>≠x<j>, in Definition IV.6 prevents any basic execution cycle which leaves the instantaneous configuration completely unchanged from being considered a DPMS computation step. Since the formal semantics for all other DYMOL statements dictate, at the very least, the modification of a process state location counter, while initialization of a total selection causes the addition of a new keyworded multiset to a process state tuple, only a process' unsuccessful attempt to receive a message is actually excluded from being a computation step by this inequality.

Models and Computations in DPMS

We have now developed a set of facilities providing all the capabilities necessary for the modelling of parallel systems with dynamic structure. Process templates, defined by abstract programs in the Dynamic Modelling Language, can be used to describe potential PSDS components. Instantaneous configurations serve as a formal representation for the various possible configurations which a PSDS may exhibit over the course of its operation. Computation steps, as determined by the formal semantics of the Dynamic Modelling Language, allow for descriptions of the changes in PSDS configurations corresponding to the operation of the system. Together these compose the Dynamic Process Modelling Scheme.

Given this modelling scheme, a *model* in the scheme consists of a set of process templates, an initial instantaneous configuration and (optionally) a set of terminal instantaneous configurations. More formally, we may define a DPMS model as follows:

> *Definition IV.7*: A Dynamic Process Modelling Scheme *model* M=(P,x<0>,T) where:
> P is a set of process templates defined by abstract programs in the Dynamic Modelling Language,
> x<0> is an instantaneous configuration, called the *initial instantaneous configuration*, and
> T is an optional set of instantaneous configurations, called the *terminal instantaneous configurations set*.

Thus, to model a parallel system with dynamic structure using the Dynamic Process Modelling Scheme, one describes the set of potentially active components, i.e., processes, as DYMOL process templates, specifies an initial system configuration and perhaps specifies any terminal configurations of interest. An example of interesting terminal configurations might be the set of configurations corresponding to system deadlock in a complex software system; a related instance is considered in the example presented in the next section. Notice that the specification of an initial instantaneous configuration, x<0>, requires that a set of process identifiers be given, both for use as indices of the configuration matrix, C<0>, and, concomitantly, as elements of the active process set, A<0>. Therefore, the process identifiers assigned to those processes which are initially instantiated in the modelled system are determined by the modeller. This is, however, the full extent of the modeller's control over process identifiers; all subsequently-created processes will be assigned new, unique process identifiers, distinct from those in A<0> but otherwise in no way influenced by the modeller of the system.

As indicated by Definition IV.7, a DPMS model serves to depict a parallel system with dynamic structure in terms of its potential processes and their possible behavior, its initial configuration and perhaps a set of conceivable

future configurations whose occurrence would be of interest. Such a static description of a PSDS could, in itself, prove useful in applications such as the design of complex, dynamically-structured software systems, providing a complete, yet abstract, representation of some particular system of interest. Generally, however, this type of static description of a PSDS is of less interest than a description which is directed toward the possible behaviors of the system. In the Dynamic Process Modelling Scheme this latter sort of PSDS description is stated in terms of computations.

A DPMS computation is simply a sequence of computation steps and therefore a sequence of instantaneous configurations, possibly constrained by the terminal instantaneous configuration set, T. In order to represent the sequencing of computations we first define a continuation operation as follows:

> Definition IV.8: The continuation operation, denoted by the continuation operator '•', maps pairs of strings of instantaneous configurations into strings of instantaneous configurations according to the following rule:
> x<i>x<j>•x<j>x<k>=x<i>x<j>x<k>
> if the two instantaneous configurations x<j> occurring immediately adjacent to the continuation operator are identical. If the operands are such that the instantaneous configurations occurring immediately adjacent to the continuation operator are not identical, the continuation operation is undefined with respect to those operands.

We may then make the following:

> Definition IV.9: A Dynamic Process Modelling Scheme computation for a given model M is denoted by z and consists of a series of computation

> steps, the first of which begins with the initial instantaneous configuration of M and none of which includes an element of M's terminal instantaneous configuration set, T (if one was specified), except that the final step of the computation may end with such an element. Formally stated:
>
> $$z=y\langle 0,1\rangle \bullet y\langle 1,2\rangle \bullet \ldots \bullet y\langle t-1,t\rangle$$
> $$=x\langle 0\rangle x\langle 1\rangle x\langle 2\rangle \ldots x\langle t-1\rangle x\langle t\rangle$$
>
> where $x\langle i\rangle$ is not an element of $T\langle M\rangle$ for $0 \leq i < t$.

Finally, we define the DPMS representation for all the possible behaviors of a modelled PSDS:

> Definition IV.10: A Dynamic Process Modelling Scheme computation set for a given model M is denoted by $Z\langle M\rangle$ and
>
> $$Z\langle M\rangle = \{z \mid z \text{ is a computation for } M\}.$$

Clearly it is the contents and properties of $Z\langle M\rangle$ which are usually of interest to the modeller of a parallel system with dynamic structure. Unfortunately, the computation sets of many interesting models are infinitely large, and it can be shown that not all questions regarding the computation set of a given DPMS model are decidable. These issues and their implications are addressed in the next chapter of this dissertation. In a subsequent chapter we demonstrate the existence of an effective procedure for the derivation from DPMS models of certain other characterizations of PSDS behavior which are applicable to an interesting subclass of these systems. We conclude the present chapter, however, with an example of DPMS modelling which illustrates both the modelling scheme itself and its potential utility as a tool for aiding in the design and analysis of complex, dynamically-structured software systems.

A DPMS Example

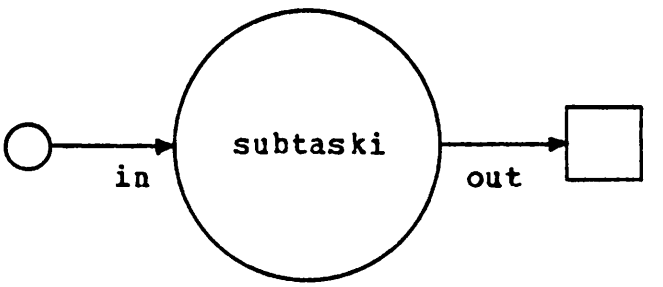
Our presentation of the Dynamic Process Modelling Scheme completed, we now consider an example of its use which may help to clarify some of the concepts introduced in the foregoing pages. The example is also meant to be suggestive of the useful role which DPMS can fulfill as an aid to the design and analysis of dynamically-structured software systems.

In this example the Dynamic Process Modelling Scheme is used to model a system consisting of a set of subtask processes which are created and destroyed by a central scheduler. These subtasks, in order to perform their function, all must utilize a common, shared resource requiring exclusive access, i.e., at most one subtask may be accessing the resource at any given time. An example of such a situation might be an on-line reservation system composed of a number of reservation-making subtasks accessing a common database, where a scheduler varies the number of active subtasks in response to changes in system load. The DPMS model described here could represent a preliminary, high-level design for a dynamically-structured software system implementing such a reservation system. It could thereby provide the software system's designer with a rigorous, although abstract, description of the proposed system, focusing on the process interaction aspects of the

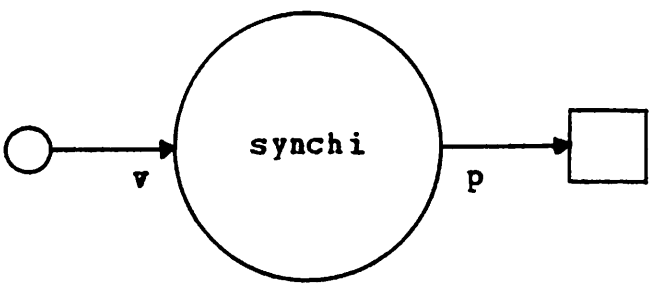
design. Moreover, being based upon a well-defined formal structure, the model allows for a certain amount of analysis of the design at an early stage in its evolution, and therefore may lead to early detection and correction of errors. Thus, design flaws may be discovered and repaired in a timely and economical fashion with the aid of DPMS. In the absence of this abstract modelling capability, such flaws often persist through many stages of design elaboration only to manifest themselves later when isolating them is more difficult and correcting them more costly.

A DPMS model for the system described above employs three process classes, whose templates are displayed in Figures IV-3 and IV-4. The process classes representing the subtasks and the process which will synchronize their access to the shared resource carry the process identifiers 'subtask' and 'synch', respectively, in Figure IV-3. The synchronizer's function is essentially to simulate a binary semaphore, as its port names 'p' and 'v' might suggest. The function of the subtasks is, however, not fully specified by their DYMOL description and is therefore left open to some interpretation depending upon the particular system being modelled. This is in keeping with the DPMS abstraction of process computation and the scheme's focus upon process interaction. Typically, the subtasks would perform any activities not requiring use of the shared resource either prior to the execution of statement st2 or after executing

```
subtask:
  st1:  DO FOREVER
          BEGIN
  st2:      RECEIVE in;
  st3:      SEND out
          END.
```




```
synch:
  sy1:  DO FOREVER
          BEGIN
  sy2:      RECEIVE v;
  sy3:      SEND p
          END.
```



Synchronizer and Subtask Process Templates

Fig. IV-3

statement st3. The RECEIVE instruction would then be interpreted as a request for access to the shared resource, with the SEND signifying the subtask's relinquishing of that access. Activity (not explicitly represented by the DYMOL template[1]) occurring between the execution of statements st2 and st3 would thus constitute a 'critical region' in which a subtask could access the shared resource and during which it could presumably expect to have exclusive control of the resource. In the reservation system interpretation of this model, the activity prior to statement st2 might involve the subtask's dialogue with an agent and the decoding of a reservation request, with the post-st3 activity consisting of a confirmation or denial of the requested reservation, and the critical region between st2 and st3 encapsulating the requisite database querying and updating.

It should be observed that refinements to a software designer's conception of the subtasks' function could easily be accompanied by elaborations of the DYMOL description, e.g., addition of explicit representation of the communication with agents and database(s) in the case of the reservation system interpretation. The Dynamic Process Modelling Scheme is, therefore, well-suited for use in

---

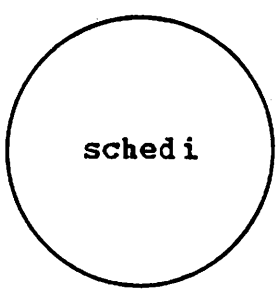
[1]It is sometimes desirable to model the activity in such a critical region somewhat more explicitly, using buffer assignments and SENDs to represent the critical computations. For our purposes here, the complete abstraction of Figure IV-3 is adequate.

design methods based upon iterative enhancement, much as Riddle's program process scheme has been shown to be useful in the design of non-dynamically structured concurrent software ([Ridd73], [Sang77], [Ridd77a]).

The final process class used in our example model is described by the scheduler template of Figure IV-4. The scheduler alternately creates new subtasks, connecting them to the synchronizer and recording their process identifiers for future use, and destroys subtasks, all according to some criteria based upon computation internal to the scheduler process which is depicted abstractly here by the WHILE INTERNAL TEST construct. Under the reservation system interpretation, this internal computation might be based upon periodic sampling of the number of busy telephone lines or some other measure of the system's activity level.

Given these three process templates, the model M may be represented formally and graphically as shown in Figure IV-5. The initial instantaneous configuration includes a scheduler process and a synchronizer process, both assigned process identifiers as part of the description of the model. The link associated with port 'p' of process synch1 initially contains a message of class 'sem' whose receipt will grant a subtask process access to the shared resource (which is not itself explicitly represented in the model). The significance of the set of terminal configurations, T,

```
sched:
  sc1:  WHILE INTERNAL TEST DO
          BEGIN
  sc2:      WHILE INTERNAL TEST DO
              BEGIN
  sc3:          CREATE subtask tvar;
  sc4:          ESTABLISH tvar.out synch1.v;
  sc5:          ESTABLISH synch1.p tvar.in;
  sc6:          /subs/ := tvar
              END;
  sc7:      WHILE INTERNAL TEST DO
              BEGIN
  sc8:          FOR SOME tvar :- /subs/ DO
                  BEGIN
  sc9:              DESTROY tvar
                  END
              END
          END.
```



Scheduler Process Template

Fig. IV-4

M = (P, x<0>, T)

where:

P = {sched, synch, subtask}

x<0> = (C<0>, A<0>, Q<0>, L<0>)

T = {(C,A,Q,L) | ∀i q[subtaski]=(st2,-), q[synch1]=(sy2,-), L=(∅,...,∅)}

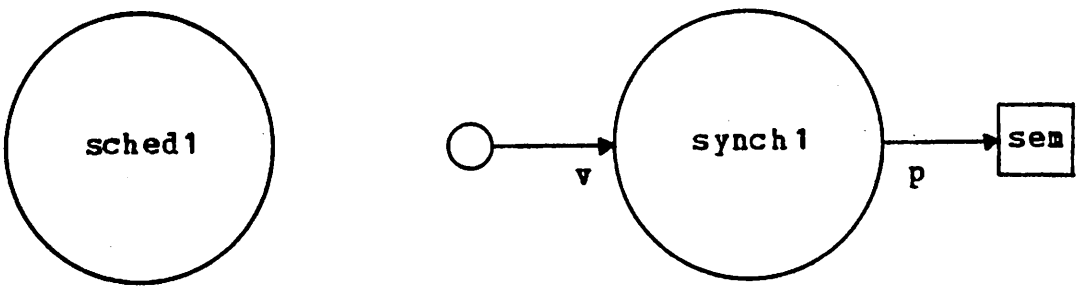
with:

C<0> =

	sched1	synch1
sched1	∅	∅
synch1	∅	∅

A<0> = {sched1, synch1} L<0> = ((<sem>))

Q<0> = ((sc1, tvar=∅, /subs/=∅, ∅), (sy1, ∅))



The Example Model M

Fig. IV-5

described in Figure IV-5 will become apparent shortly. T's representation in terms of a partial specification is typical of the usage of this feature of the Dynamic Process Modelling Scheme; those parts of the terminal configurations not explicitly bound by this specification, including those represented by '-' in the process state tuples, are free to assume any value within their range.

The model M can be used for some analysis of a proposed software system design as well as for description. The designer might, for instance, determine that one possible instantaneous configuration, x<i>, which M might attain is that shown formally in Figure IV-6 and graphically in Figure IV-7[1]. Three subtask processes currently exist in the system depicted by x<i>, each connected to the synchronizer process, synch1. One of these, subtask7, is currently in its critical region and will soon relinquish access to the shared resource by returning the sem message to the synchronizer, as indicated by its process state. The others, according to their process states, are currently awaiting the opportunity to access the shared resource. Consideration of the computation steps possible from instantaneous configuration x<i> offers the designer some

---

[1]This can be determined by manually producing a computation leading to x<i>, as is done in detail in Appendix C. Alternatively, a facility for the automatic derivation of possible instantaneous configurations could be constructed as part of an automated design aid for dynamically-structured, concurrent software systems.

x<i> = (C<i>, A<i>, Q<i>, L<i>)

where

C<i> =

	sched1	synch1	subtask2	subtask7	subtask9
sched1	∅	∅	∅	∅	∅
synch1	∅	∅	(p,in)	(p,in)	(p,in)
subtask2	∅	(out,v)	∅	∅	∅
subtask7	∅	(out,v)	∅	∅	∅
subtask9	∅	(out,v)	∅	∅	∅

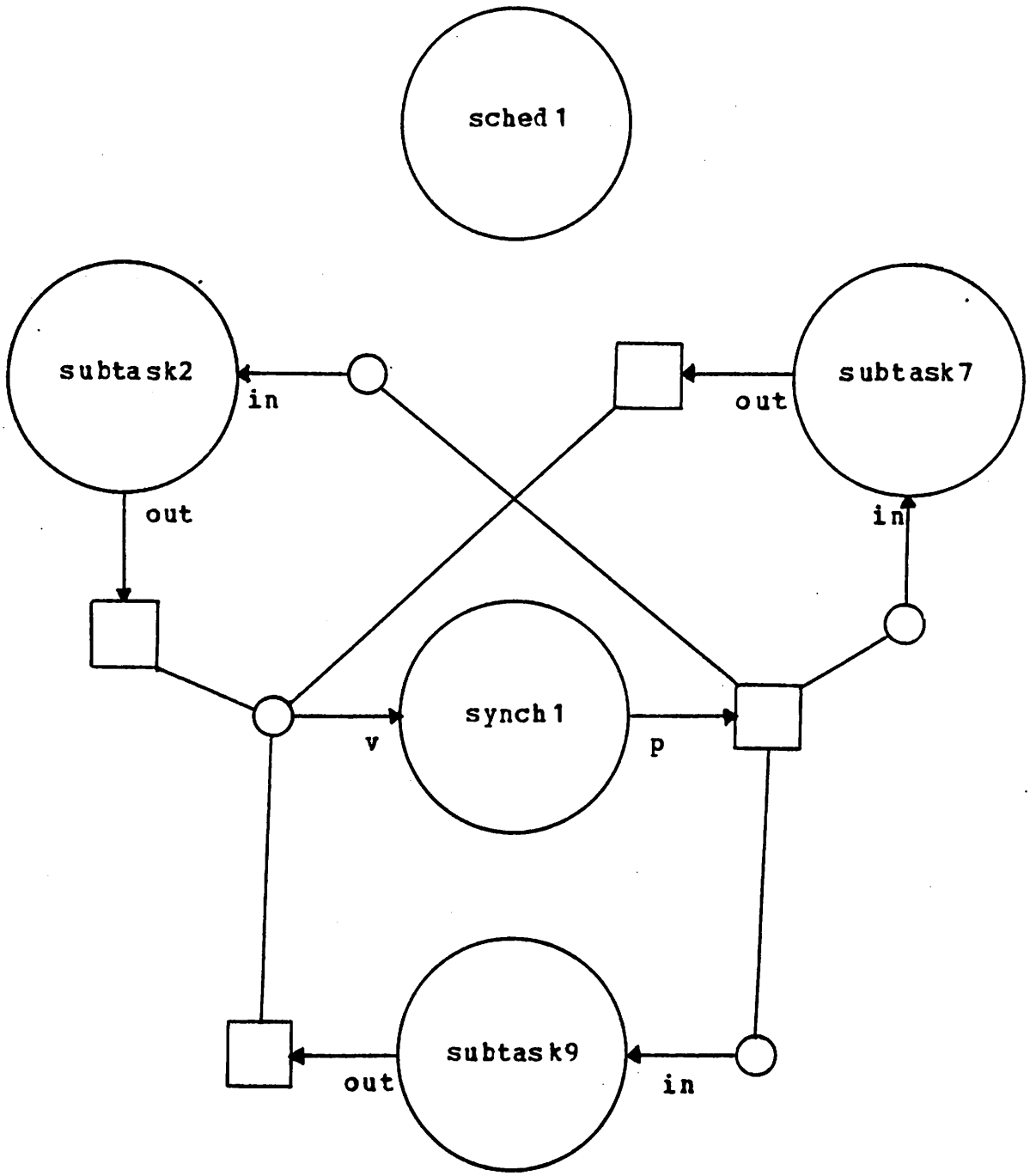
A<i> = {sched1, synch1, subtask2, subtask7, subtask9}

Q<i> = ((sc9,/subs/=<subtask2,subtask9>,tvar=<subtask7>,∅),
(sy2,sem),(st2,sem),(st3,sem),(st2,∅))

L<i> = (∅,∅,∅,∅)

Formal Representation of Instantaneous Configuration x<i>

Fig. IV-6



Graphical Representation of Instantaneous Configuration x<i>

Fig. IV-7

important information regarding the design modelled by M.

One possible computation step $y\langle i,i+1\rangle = x\langle i\rangle x\langle i+1\rangle$ is shown in Figure IV-8. Only the process state tuple q[subtask7] in the complete process state, Q, and the link state associated with subtask7's outbound port in the complete link state, L, are altered as a result of this computation step, which can be interpreted as the surrendering of control of the shared resource by subtask7. Some other subtask can thus be granted access to the resource once the synchronizer makes the sem message available for reception by one of them. In all likelihood, this orderly continuation of system activity is in accordance with the designer's intentions for system behavior.

Another possible computation step $y'\langle i,i+1\rangle = x\langle i\rangle x'\langle i+1\rangle$ is shown in Figure IV-9. In this instance, the scheduler process has destroyed subtask7, as evidenced by the appropriate changes to C, A, Q, and L. The instantaneous configuration, $x'\langle i+1\rangle$, resulting from this computation step is an element of the set of terminal configurations, T, for the model M. It can now be seen that T corresponds to the set of conceivable system configurations which yield <u>subtask deadlock</u>, as demonstrated by the present instance. While the overall system is still capable of performing additional computation steps, which may include the creation and

y<i,i+1> = x<i>x<i+1>

where

x<i+1> = (C<i>, A<i>, Q<i+1>, L<i+1>)

Q<i+1> = ((sc9,/subs/=<subtask2,subtask9>,
tvar=<subtask7>,Ø),(sy2,sem),(st2,sem),
(st1,sem),(st2,Ø))

L<i+1> = (Ø,Ø,(<sem>),Ø)

A Possible Computation Step from x<i>

Fig. IV-8

y'<i,i+1> = x<i>x'<i+1>

where

x'<i+1> = (C'<i+1>, A'<i+1>, Q'<i+1>, L'<i+1>)

with

C'<i+1> =

	sched1	synch1	subtask2	subtask9
sched1	∅	∅	∅	∅
synch1	∅	∅	(p,in)	(p,in)
subtask2	∅	(out,v)	∅	∅
subtask9	∅	(out,v)	∅	∅

A'<i+1> = {sched1, synch1, subtask2, subtask9}

Q'<i+1> = ((sc9,/subs/=<subtask2,subtask9>,
tvar=<subtask7>,∅),(sy2,sem),(st2,sem),(st2,∅))

L'<i+1> = (∅,∅,∅,∅)

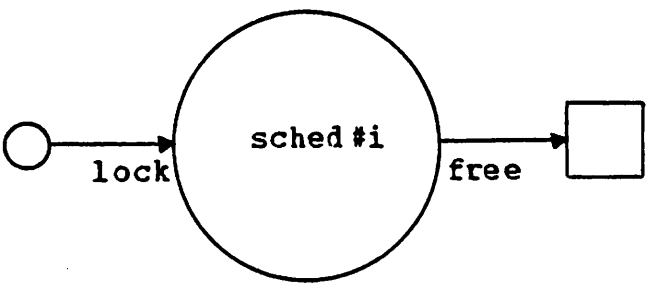
Another Possible Computation Step from x<i>

Fig. IV-9

destruction of subtasks, no subtask process can possibly progress after x'<i+1> since the sem message has disappeared with the destruction of subtask7 and none of M's processes is capable of generating messages. Thus, all subtask processes in the system will henceforth wait indefinitely for permission to access the shared resource, suspended at the RECEIVE instruction labelled 'st2'. The appearance of a computation resulting in subtask deadlock would most likely indicate to a software system designer that the design modelled by M is incorrect. This example, then, illustrates how the analysis made possible by the formal structure of the Dynamic Process Modelling Scheme can be beneficially applied to proposed designs for dynamically-structured, concurrent software systems.

We conclude this example and this chapter by presenting a revised version, M', of the example model which is demonstrably free from the possibility of subtask deadlock. The revised model uses the process classes 'subtask' and 'synch' as defined previously, but employs a modified scheduler whose process template 'sched#' appears in Figure IV-10. The modified scheduler differs from the previous version only in the addition of the statements labelled 's#1' and 's#2', which effect the necessary synchronization of subtask destruction and the subtask critical regions. With appropriate linkage between the scheduler and synchronizer, initially established within M' as shown in

```
sched#:
  sc1:  WHILE INTERNAL TEST DO
          BEGIN
  sc2:      WHILE INTERNAL TEST DO
              BEGIN
  sc3:          CREATE subtask tvar;
  sc4:          ESTABLISH tvar.out synch1.v;
  sc5:          ESTABLISH synch1.p tvar.in;
  sc6:          /subs/ := tvar
              END;
  sc7:      WHILE INTERNAL TEST DO
              BEGIN
  sc8:          FOR SOME tvar :- /subs/ DO
                  BEGIN
  s#1:              RECEIVE lock;
  sc9:              DESTROY tvar;
  s#2:              SEND free
                  END
              END
          END.
```



Revised Scheduler Process Template

Fig. IV-10

Figure IV-11, the modified system will not permit the destruction of a subtask while that subtask is in its critical region, thus preventing subtask deadlock.

The assertion that M' is free from the possibility of subtask deadlock may be more formally stated as:

> Assertion IV.1: There is no computation z in Z<M'> such that z=x<0>...x<t> and x<t> is an element of T'.

The truth of Assertion IV.1 may be seen from the following argument:

Clearly x<0> is not an element of T'. Now suppose that there were such a computation z in Z<M'>. Since the initial complete link state of M' is L'<0>=(∅,(<sem>) and the complete link state of x<t> is L'<t>=(∅,...,∅) it must be the case that some process executed a RECEIVE instruction during computation z without subsequently executing a SEND. But this can only be true of a currently instantiated process if lc[sched#2]=sc9 or s#2, if lc[synch1]=sy3 or if lc[subtaski]=st3 for some i. Since Q'<t> precludes all of these possibilities, and only processes of class 'subtask' are ever destroyed in M', it must be the case that some subtask, say subtaskj, was destroyed in z with lc[subtaskj]=st3. But since RECEIVEs and SENDs are strictly paired in the process templates of M' there can never be more than one sem message in any complete link state of M'. Hence, there can never be two executions of RECEIVE instructions in M' without an intervening SEND. Now in

M' = (P',x'<0>,T')

where:

P' = {sched#, synch, subtask}

x'<0> = (C'<0>, A'<0>, Q'<0>, L'<0>)

T' = [(C,A,Q,I) | ∀i q[subtaski]=(st2,-),
q[sched#2]=(s#1,-,-,-),
q[synch1]=(sy2,-), L=(∅,...,∅)]

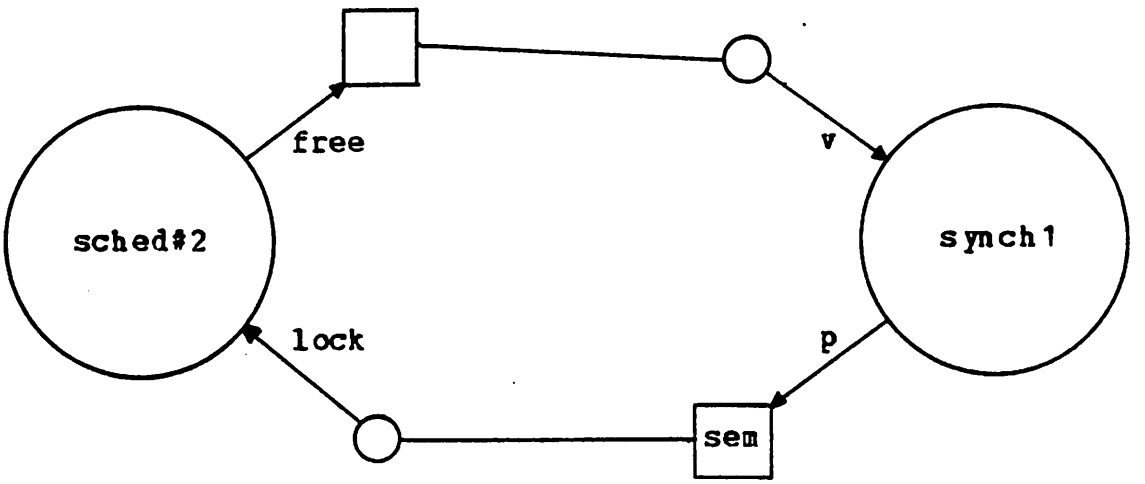
with:

C'<0> =

	sched#2	synch1
sched#2	∅	(free,v)
synch1	(p,lock)	∅

A'<0> = {sched#2,synch1} L'<0> = (∅,(<sem>))

Q'<0> = ((sc1,tvar=∅,/subs/=∅,∅), (sy1,∅))



The Revised Example Model M'

Fig. IV-11

order to destroy subtaskj, sched#2 would have first been required to execute the RECEIVE instruction at s#1, and therefore lc[subtaskj]=st3 would imply that two RECEIVEs had been executed without an intervening SEND. This being impossible in M', we may conclude that no such z exists in Z<M'>, as asserted.

The capability for rigorously demonstrating the truth of assertions regarding properties of DPMS models, illustrated by the preceding proof of Assertion IV.1, suggests yet another facet of the potential utility of the Dynamic Process Modelling Scheme to designers of dynamically-structured, concurrent software systems. Not only may the scheme be employed to describe a system's design and to uncover possible flaws therein, but it can also serve as a vehicle for formal verification of properties ascribed to the design. While no general procedure can be defined for attaining proofs of such purported properties, as we show in the next chapter, the ability to rigorously confirm particular properties in particular cases can nonetheless provide a powerful tool for software system designers using the Dynamic Process Modelling Scheme.

CHAPTER V

DECIDABILITY PROPERTIES OF THE DYNAMIC PROCESS MODELLING SCHEME

We now turn to an investigation of the properties and capabilities of DPMS and of the parallel systems with dynamic structure which the scheme can be used to represent. That investigation constitutes the remainder of this dissertation.

In this chapter we begin by examining certain decidability properties of the complete Dynamic Process Modelling Scheme. We demonstrate two different ways in which Turing machines may be simulated using the modelling scheme and derive several undecidability results based upon these constructions. The ramifications of these results are then explored before we proceed, in the next chapter, to discuss a PSDS subclass, and DPMS models for it, yielding more favorable decidability results.

A DPMS Turing Machine Construction

In this section we show that it is possible to simulate an arbitrary Turing machine using a Dynamic Process Modelling Scheme model. The construction presented here will then serve as a basis for proving the undecidability of several questions concerning DPMS models. In essence, the remainder of this section constitutes a proof of the following:

> Theorem V.1: For any given Turing machine, TM, there exists a DPMS model, M[TM], such that the computations of M[TM] precisely simulate the possible computations of the given Turing machine.

As this theorem indicates, for a given Turing machine TM (represented in quintuple notation as described below) we can construct a DPMS model M[TM] which simulates the computations of the Turing machine. We say that the resulting simulation is "precise" because, at every step in any computation of the simulating model, there is a well-defined correspondence between the model's instantaneous configuration and some instantaneous description (i.e., machine state and tape marking) of the simulated Turing machine. The simulation uses a set of processes to represent the Turing machine's tape, with one pair of processes corresponding to each square of the simulated tape. A message stored in the buffer of one of these

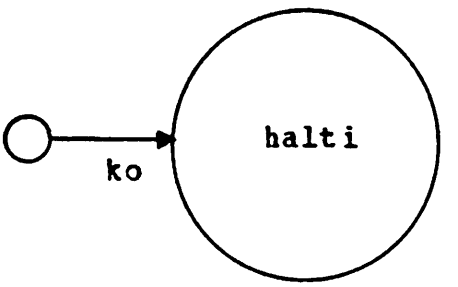
processes represents the current marking of the tape square to which the pair of processes corresponds. The simulated tape initially consists of only one blank square and is dynamically extended to whatever length is required during the simulated Turing machine's computation. The simulated Turing machine's current state is also represented by a message stored in the buffer of a designated process in M[TM]. The position and movement of the simulated machine's tape read/write head are represented by particular patterns of message transmission in the DPMS model. Thus, the buffer contents and process states of certain processes in M[TM] encode the current instantaneous description of the simulated Turing machine at any point during the simulation. The computations of the DPMS model, therefore, directly represent the possible computations of the simulated Turing machine through this encoding.

Various equivalent formulations of the Turing machine have been developed (see for example [Mins67]) since Turing's original definition appeared in 1936 [Turi36]. For use in this construction, we adopt the reasonably standard version which specifies the machine's computational possibilities in terms of quintuples and employs a one-way infinite tape memory. The quintuples are of the form '(q1,x1,q2,x2,d)' where q1 and x1 are the machine state and scanned tape square marking, respectively, which are the necessary preconditions to the quintuple's application.

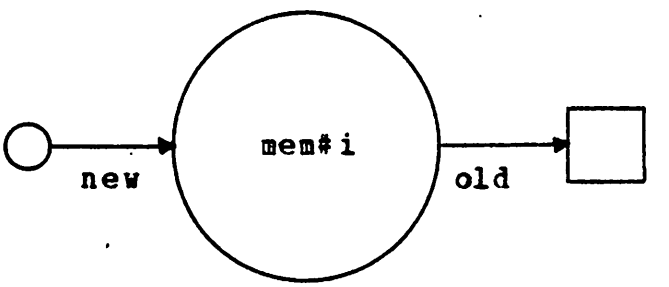
Then q2 is the new machine state, x2 the new mark placed in the scanned tape square and d the direction, either left or right, of movement for the machine's read/write head which result from the application of the quintuple. We will assume that the tape extends infinitely to the right and is initially blank, neither assumption entailing any loss of generality. We also subscribe to the convention that the Turing machine halts at any point during its computation when no quintuple's q1 and x1 match the current machine state and scanned tape square marking.

The construction begins with the definition of process templates for the various components of the simulated Turing machine. The first two of these are the simple templates shown in Figure V-1. The process class 'halt' is used to explicitly represent the halting of the simulated Turing machine. Should a process of this class ever actually receive a message through its ko port, i.e., if its request for a message is ever fulfilled, then the simulated Turing machine's computation would terminate and conversely. The process class 'mem#' is used as a repository for information by both the tape squares and the simulated Turing machine's control, holding tape markings for the former and machine states for the latter. A process of this class receives a value and then simply sends it again, making the value available for later retrieval by the process using that memory. The uses of both of these process classes will

```
halt: ht1: RECEIVE ko.
```




```
mem#: m1: DO FOREVER
          BEGIN
      m2:   RECEIVE new;
      m3:   SEND old
          END.
```



Halt and Memory Process Templates for
Turing Machine Construction

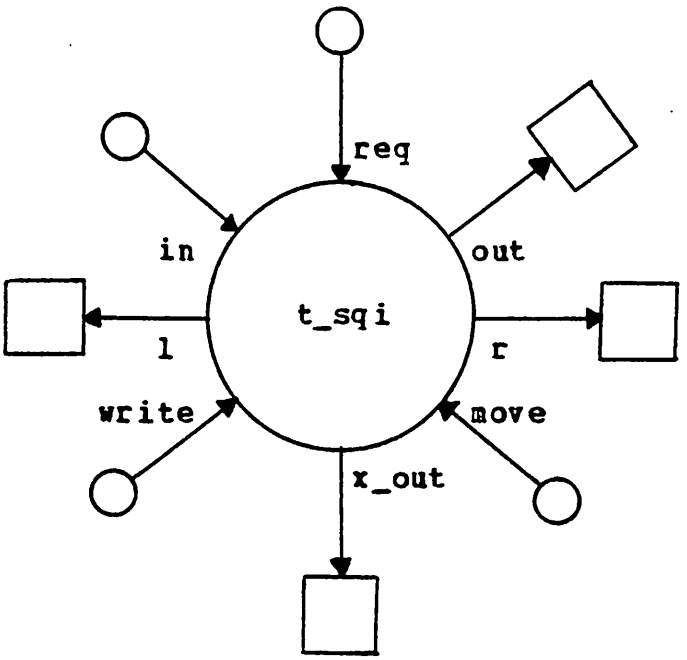
Fig. V-1

become evident in succeeding pages.

The process template used to represent the individual squares of the DPMS-simulated Turing machine's memory tape is perhaps the most interesting part of the construction and the most significant aspect of the approach employed in our Turing machine representation. This template, whose process class identifier is 't_sq', is shown in Figure V-2. Its DYMOL program may conveniently be viewed as consisting of two sections. The first section, encompassing the statements labelled sq1 through sq12, is a prologue which serves to initialize the square and to extend the tape by one square whenever its current rightmost square is read for the first time. This prologue is executed only once by each instantiation of process class t_sq. The remainder of the tape square program, statements sq13 through sq22, are repeated indefinitely and represent the protocol for reading and writing of the tape square and also for the conceptual movement of the simulated Turing machine's read/write head.

As will be seen shortly, the initial configuration of a DPMS Turing machine simulation includes just one tape square process, connected to a process of class 'mem#' via its ports labelled 'in' and 'out'. The reading of this initial square by the simulated machine begins upon the reception of a message through the square's inbound 'req' port. Reception of that message, i.e., completion of statement

```
t_sq: BEGIN
 sq1:   RECEIVE req;
 sq2:   SET BUFFER := blank;
 sq3:   SEND out;
 sq4:   CREATE t_sq new_sq;
 sq5:   CREATE mem# new_mem;
 sq6:   ESTABLISH new_sq.out new_mem.new;
 sq7:   ESTABLISH new_mem.old new_sq.in;
 sq8:   ESTABLISH ME.r new_sq.req;
 sq9:   ESTABLISH new_sq.l ME.req;
sq10:   ESTABLISH tm1.move new_sq.move;
sq11:   ESTABLISH tm1.wrt new_sq.write;
sq12:   ESTABLISH new_sq.x_out tm1.rd;
sq13:   DO FOREVER
          BEGIN
sq14:       RECEIVE in;
sq15:       SEND x_out;
sq16:       RECEIVE write;
sq17:       SEND out;
sq18:       RECEIVE move;
sq19:       IF BUFFER=left THEN
sq20:         SEND l
            ELSE
sq21:         SEND r;
sq22:       RECEIVE req
          END
      END.
```



The Tape Square Process Template

Fig. V-2

sq1, then triggers the prologue which first, at statements sq2 and sq3, initializes the memory process which serves as a repository for the square's marking by sending a message of class 'blank' to that memory process[1]. The prologue then causes the creation of a new tape square and memory (sq4 and sq5), connects the two new processes together in the appropriate fashion (sq6 and sq7), and connects (sq8) the new square's 'req' port to the outbound port labelled 'r' of the previously rightmost square of the tape, namely the currently executing process itself. Similarly, at statement sq9, the previously rightmost square's 'req' port is connected to the new square's outbound port labelled '1'. It is these two ESTABLISH operations which serve to make the newly-created square the rightmost of the tape and which encode the necessary information to allow the conceptual right and left movement of the simulated Turing machine's read/write head. Finally, statements sq10, sq11 and sq12 connect the new rightmost tape square to the control process, tm1, in such a way that the new square may be read, written and directed to move the read/write head. Thus, upon completion of the prologue's execution, the square being read has, in effect, been initialized to be blank and the tape has been extended so that there is always a square to the right of the current read/write head position. Since

---

[1]We make the usual assumption that 'blank' is a distinct marking from any of the other possible values for tape markings, an assumption enforced by a simple renaming of markings when necessary.

this same prologue is performed whenever a rightmost square of the tape is read for the first time, the DPMS-simulated tape is appropriately extended, one (initially blank) square at a time, to whatever length is required during the course of the Turing machine's computation.

Having once extended the tape by creating and connecting a new square, the tape square process proceeds into the infinite iteration which makes up the second section of its DYMOL description. This begins with the square's responding to the read request lodged (generally indirectly, as will be seen below) by the controller process, tm1. It accomplishes this by first executing the RECEIVE instruction at statement sq14, thus retrieving the current marking stored in its associated memory process, then sending the retrieved value through its outbound port x_out. Having thus responded to the read request, the square may then execute statements sq16 and sq17, resulting in the updating of its marking as recorded by the associated memory process. This updating completed, the square awaits instruction from tm1 regarding movement of the read/write head. When this instruction has been provided, i.e., the RECEIVE statement at sq18 has been completed, the tape square activates its immediately adjacent neighbor to either the left or the right, according to the instruction sent by tm1 (statements sq19, sq20 and sq21). The message sending which ensues from this conditional statement allows the

appropriate neighboring tape square process to complete a RECEIVE operation on which it had been suspended and begin executing statements which will lead to the delivery of a message representing its current marking to the controller, tm1. It should be noted that this pattern of activity assures that one tape square's marking is updated before the next square may be read. Meanwhile, the execution of either of sq20 or sq21 brings the currently executing tape square process to the RECEIVE instruction, sq22, where it will be suspended until re-activated by one of its neighbors in response to a read/write head movement instruction from tm1. Upon re-activation, the square would repeat this second section of its DYMOL program, thereby being read and written and causing the (conceptual) movement of the read/write head. This procedure is repeated until the controller either halts or attempts to move the read/write head position to the left of the initial tape square process. Either of these causes the transmission of a message to a halt process and termination of execution in the DPMS Turing machine model.

The three process templates defined above may be included, unchanged, in any DPMS Turing machine model. The final template required for such a model encodes the actual set of quintuples corresponding to the particular Turing machine being simulated and hence will have a somewhat different DYMOL program for each such machine. This process

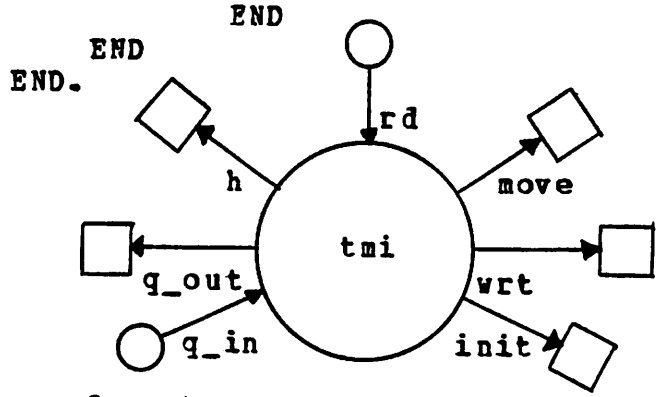
template, representing the machine's controller, has the process class identifier 'tm' and its DYMOL specification is of the form indicated in Figure V-3. Each pair of machine states and tape markings (q1,x1) appearing as a precondition in a quintuple of the modelled Turing machine corresponds to a pair of conditional statements in the controller's DYMOL template, with the appropriate (q2,x2,d) triple of resultant Turing machine actions being represented by a set of buffer assignments and SEND operations. It may easily be verified that a Turing machine specified by n quintuples will thus produce a DPMS model whose tm process template contains $7n^2+3n+6$ labelled statements.

The controller process in the DPMS model of any non-trivial Turing machine begins by sending a message to the 'req' port of the machine model's initial tape square[1], at statements tm1 and tm2. This message leads to the transmission by the tape square process of a message representing its current marking, which can then be received by the controller through its 'rd' port at some future time (e.g., tm6). This initial request for a tape read represents the only time that the controller will directly

---

[1]Of course the message is actually merely sent out through the specified outbound port, init, into the associated link. Here, as elsewhere in this chapter, we say that the message is sent to a particular inbound port meaning that the designated port is the only one which both has a currently pending request for message and is connected to the specified link.

```
  tm:  BEGIN
 tm1:    SET BUFFER := start;
 tm2:    SEND init;
 tm3:    DO FOREVER
           BEGIN
 tm4:        RECEIVE q_in;
 tm5:        IF BUFFER=qi THEN
               BEGIN
 tm6:            RECEIVE rd;
 tm7:            IF BUFFER=xi THEN
                   BEGIN
 tm8:                SET BUFFER := qj;
 tm9:                SEND q_out;
tm10:                SET BUFFER := xj;
tm11:                SEND wrt;
tm12:                SET BUFFER := left;
tm13:                SEND move
                   END
                  ELSE
 tm14:             IF BUFFER=xk THEN
                              .
                              .
                              .
                    ELSE
                     BEGIN
 tm98:                 SET BUFFER := left;
 tm99:                 SEND h
                     END
               END
              ELSE
tm100:         IF BUFFER=qk THEN
                              .
                              .
                              .
                ELSE
                  BEGIN
tm998:              SET BUFFER := left;
tm999:              SEND h
                  END
        END
END.
```



Schema for the Controller Process Template

Fig. V-3

send a message to the req port of any tape square process. All subsequent tape reads result from tape movement instructions sent by the controller which lead to a message transmission from the currently scanned tape square to the req port of one of its neighbors.

Following the sending of this initial read request message, the controller process begins the infinite iteration which corresponds to successive interpretation of the quintuples specifying allowable steps in the particular Turing machine's computation. First the RECEIVE instruction at tm4 is executed, thereby retrieving a message representing the current machine state. Then a series of cascaded conditionals, such as those at statements tm5 and tm100 in Figure V-3, are used to select an appropriate set of possible actions according to the value of that message. Within each such conditional's embedded statement is a RECEIVE statement retrieving a message representing a tape square marking (e.g., tm6), followed by another series of cascaded conditionals, such as tm7 and tm14 in Figure V-3. These latter conditionals select the particular set of actions appropriate to the current machine state and tape marking, with their embedded statements (e.g., tm8 through tm13) leading to the occurrence of those actions. If the current machine state and tape marking satisfy the precondition of no quintuple in the modelled Turing machine's specification, then a message will be sent to the

halt process either by statements such as tm98 and tm99 (if the current machine state matched some quintuple but the tape marking did not) or by statements tm998 and tm999 (if the current machine state did not match any quintuple). Otherwise, three successive pairs of buffer assignments and SEND instructions (e.g., tm8 through tm13) will (1) update the current machine state (tm8 and tm9), (2) update the marking of the currently scanned tape square (tm10 and tm11) and (3) direct a (conceptual) left or right movement of the read/write head (tm12 and tm13).

The operation of the controller process in a DPMS-modelled Turing machine may be illustrated by the following example based upon the template of Figure V-3. Suppose that (qi,xi,qj,xj,left) were one quintuple in the specification of the modelled Turing machine. Suppose further that the memory process connected to the controller's q_in port has most recently sent a message of class 'qi', while the last message sent by any tape square is of class 'xi'. This represents a current machine state and tape marking pair of (qi,xi) in the simulated Turing machine. Then the RECEIVE statement at tm4 will cause the controller's buffer to attain the value qi, hence the condition in statement tm5 will evaluate to 'true'. Therefore, the RECEIVE statement at tm6 can be executed with the result that an xi message is placed into the controller's buffer. Thus the condition in tm7 will evaluate to 'true' and the buffer assignment

statement at tm8 will set the controller's buffer to contain a message of class 'qj', which is then sent to the same memory process from which the qi message was previously received. This serves to update the simulated machine state to qj. Statements tm10 and tm11 are then executed, sending a message of class 'xj' to the tape square which had sent the xi message previously, its being the only square with a RECEIVE request currently pending on an inbound port connected to the controller's wrt port. Finally, statements tm12 and tm13 are executed, causing the currently scanned tape square to send a message to the 'req' port of its left neighbor (or to the halt process if no such neighbor exists), thereby initiating a read of that tape square and, conceptually, moving the read/write head one square to the left on the modelled machine tape. Thus the new machine state, new tape marking and the direction of the read/write head movement correspond to those specified by the quintuple. The remaining cascaded conditionals at both levels (alternative tape marking possibilities and alternative machine state possibilities) will not be executed, due to the eos directives for the respective conditionals, and the controller's location counter will be set to tm3, in preparation for determining and acting upon the next applicable quintuple of the simulated Turing machine's specification.

The complete formal specification of a DPMS model,

M[TM], simulating a Turing machine is illustrated in Figure V-4, with a corresponding graphical representation in Figure V-5. The model includes the four process class templates discussed above, an initial instantaneous configuration representing the simulated machine's initial configuration and a terminal instantaneous configuration set corresponding to the possibilities for the halting of the simulated Turing machine. The initial instantaneous configuration represents a machine controller, tm1, and its associated memory process, mem#0, the latter's link state indicating that the simulated machine's initial state is q0. Also included in this initial instantaneous configuration are a single tape square, t_sq1, and its associated memory process, mem#1, along with the process halt1. Computations for M[TM], that is, elements of Z<M[TM]>, would then consist of a sequence of computation steps simulating the Turing machine's computation in the manner described above. Throughout this DPMS computation the current machine state of the simulated Turing machine will be represented by the current buffer contents of the memory process mem#0 while its current tape marking will be represented by the current buffer contents of the memory processes associated with all the tape square processes instantiated in the model. The appropriate ordering of the tape squares and their associated memory processes can be determined at any point during the computation through inspection of the configuration matrix, C, or the active process set, A.

M[TM] = (P, x<0>, T)

where

P = {halt, mem#, t_sq, tm}

x<0> = (C<0>, A<0>, Q<0>, L<0>)

T = {(C, A, Q, L) | q[halt1]=(Ø, left), q[tm1]=(tm6, -) or ... }

with

C<0> =

	tm1	t_sq1	mem#0	mem#1	halt1
tm1	Ø	{(init, req), (move, move), (wrt, write)}	(q_out, new)	Ø	(h, ko)
t_sq1	(x_out, rd)	Ø	Ø	(out, new)	(1, ko)
mem#0	(old, q_in)	Ø	Ø	Ø	Ø
mem#1	Ø	(old, in)	Ø	Ø	Ø
halt1	Ø	Ø	Ø	Ø	Ø

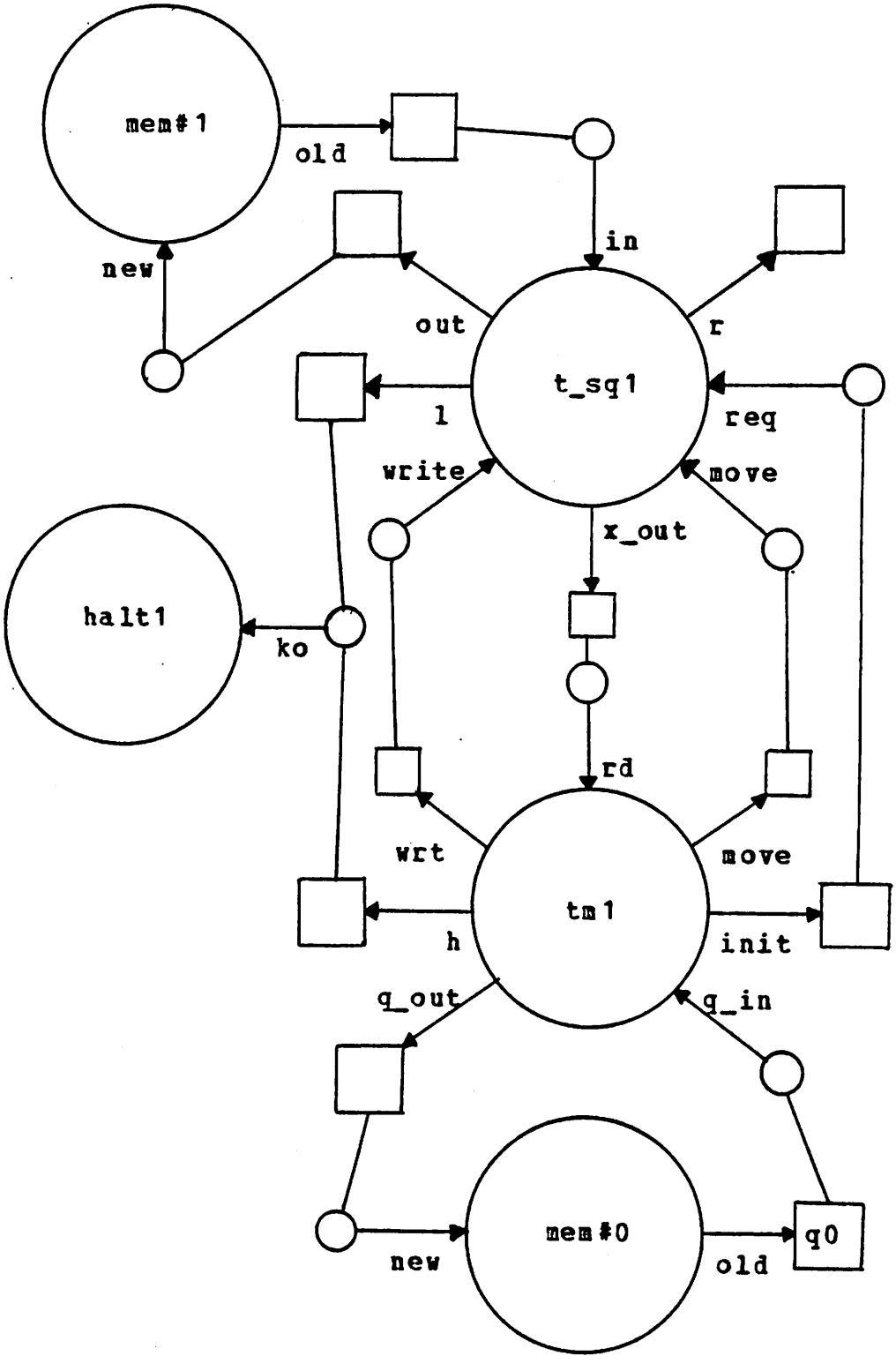
A<0> = {tm1, t_sq1, mem#0, mem#1, halt1}

Q<0> = ((tm1, Ø), (sq1, new_sq=Ø, new_mem=Ø, Ø), (m1, Ø), (m1, Ø), (ht1, Ø))

L<0> = ((Ø, Ø, Ø, Ø, Ø), (Ø, Ø, Ø, Ø), (<q0>), Ø)

DPMS Model for a Turing Machine

Fig. V-4



Initial Instantaneous Configuration of M[TM]

Fig. V-5

As we have indicated in the preceding discussion of the M[TM] process templates, any condition which would cause a given Turing machine to halt will lead to the eventual sending of a message to process halt1 within the DPMS model simulating that Turing machine. The sending of such a message always occurs in lieu of either the sending of a tape movement instruction by tm1 (if the current machine state and tape marking match no quintuple) or the sending of a message from one tape square process to another's req port (if the specified movement is to the left of the tape's initial square). In either case, no message will subsequently reach the req port of any tape square process and hence no further messages will be available to tm1's rd port. Therefore, halting of the Turing machine is represented in its DPMS model by the appearance of a message in halt1's buffer and the suspension of tm1 at a RECEIVE instruction, such as tm6, where it attempts to read a tape square[1]. Since this is exactly the situation represented by the terminal instantaneous configuration set, T, of M[TM], and since the buffer of halt1 is otherwise always empty, we see that halting Turing machine computations correspond precisely to computations in the simulating DPMS model which terminate with an instantaneous configuration from the set

---

[1]Note that since there are only a finite number of such RECEIVE instructions in a given tm template, i.e., one for each possible machine state of the simulated Turing machine, the specification of T as shown in Figure V-4 is also finite.

T. The result of the simulated machine's computation may then be determined by inspecting the current contents of the buffers of the memory processes associated with the instantiated tape squares, ordered as indicated by C or A.

Thus, construction of an arbitrary Turing machine in the Dynamic Process Modelling Scheme is merely a question of encoding the machine's (blank, right-infinite tape) quintuple representation in a process of class 'tm'. This encoding is a straightforward procedure, which is facilitated by grouping the quintuples according to their q1 entries. Then each distinct q1 yields a DYMOL conditional of the form of tm5 or tm100, while each distinct x1 for that q1 yields a conditional of the form of tm7 or tm14 along with an appropriate set of three buffer assignment - SEND statement pairs (like tm8 through tm13) encoding the q2, x2 and d of the quintuple. Having given the DYMOL specification for this process, the modeller then simply uses it as the tm process template in a model of the form shown in Figure V-4, with appropriate modification of the initial complete link state, L<0>, to reflect the simulated Turing machine's initial machine state. It is clear that this construction may be effectively carried out for any Turing machine expressed in the appropriate form. Since any arbitrary Turing machine can be expressed as one which begins with a blank, one-way infinite tape, and in quintuple form, we have thus shown that any arbitrary Turing machine

can be simulated using the Dynamic Process Modelling Scheme.

Undecidability in DPMS Models

Based upon the above-demonstrated possibility of constructing any arbitrary Turing machine using the Dynamic Process Modelling Scheme, we are now able to prove that certain questions regarding DPMS models are undecidable. The proofs for these results involve showing that decidability for the DPMS models would imply decidability for a problem known to be undecidable, namely the blank tape halting problem for Turing machines.

One question which might well be of interest to a modeller using DPMS is whether any computation of a given DPMS model M=(P,x<0>,T) has a terminal instantaneous configuration, i.e., an element of T, as its final instantaneous configuration. No effective procedure can exist for answering this question in general, as shown by the following:

> Theorem V.2: It is undecidable whether, for an arbitrary DPMS model M=(P,x<0>,T), there exists a computation of M leading to an instantaneous configuration which is an element of T.
>
> Proof: Suppose it were decidable. Then the blank tape halting problem would be decidable for any arbitrary Turing machine as follows: (1) Produce a

specification of the Turing machine using a right-infinite tape and stated in quintuples. (2) Generate a DPMS model simulating that Turing machine according to the construction described in the previous section. (3) Determine whether the resulting DPMS model admits of a computation leading to one of its terminal instantaneous configurations. The Turing machine in question then halts if and only if such a computation exists for the DPMS model so constructed. But since the blank tape halting problem for an arbitrary Turing machine is undecidable, this results in a contradiction and hence it must be undecidable whether there exists a computation leading to a terminal instantaneous configuration for an arbitrary DPMS model.

It follows from Theorem V.2 that it is not, in general, decidable whether any given instantaneous configuration will arise in the possible computations of a DPMS model. Specifically, we have:

Corollary V.3: It is undecidable whether, for an arbitrary DPMS model M=(P,x<0>,T) and an arbitrary instantaneous configuration x, any computation of M will produce x.

Proof: If this were decidable, then the procedure for deciding it could be used to decide the question proven undecidable in Theorem V.2.

For a given DPMS model, each computation produces a unique series of message transmissions, where a message transmission is said to occur each time that a message moves into or out of one of the links in the model. That is, each execution of a SEND instruction by a process whose buffer is non-empty and each successful completion of a RECEIVE instruction execution represents a message transmission. The message sequence resulting from such a series of message transmissions is a string of message types corresponding to the messages which actually moved into or out of a link with each message transmission occurring during the computation. In the M[TM] model developed above, for instance, a message sequence would begin 'start start blank ...', the first element corresponding to a SEND instruction of tm1, the second and third to a RECEIVE and SEND, respectively, performed by process t_sq1. Although the preceding results indicate that questions regarding the reachability of particular instantaneous configurations are undecidable for DPMS models, we might still hope to find effective procedures for determining whether a given message or series of messages could occur within the message sequence corresponding to some computation of a given DPMS model. However, this too turns out to be impossible, as shown by the following:

> Theorem V.4: It is undecidable whether, for an arbitrary DPMS model M=(P,x<0>,T) and an arbitrary message class, any computation of M will lead to

the transmission of a message of the specified message class.

Proof: Suppose it were decidable. Then the blank tape halting problem would be decidable for any arbitrary Turing machine as follows: (1) Produce a specification of the Turing machine using a right-infinite tape and stated in quintuples. (2) Generate a DPMS model simulating that Turing machine according to the construction described in the previous section. (3) Modify the halt process of the DPMS model such that, upon receiving a message, it would proceed to transmit a message of some class distinct from any other message class defined in the model. (4) Determine whether the resulting DPMS model admits of a computation leading to the transmission of a message of the newly-added message class. The Turing machine in question then halts if and only if such a computation exists for the DPMS model so constructed. But since the blank tape halting problem for an arbitrary Turing machine is undecidable, this results in a contradiction and hence it must be undecidable whether there exists a computation leading to the transmission of a message of a specified class for an arbitrary DPMS model.

It follows from Theorem V.4 that it is not, in general,

decidable whether any given series of messages will be transmitted during one of the possible computations of a DPMS model. Specifically, we have:

Corollary V.5: It is undecidable whether, for an arbitrary DPMS model M=(P,x<0>,T) and an arbitrary series of message classes, any computation of M will lead to a series of message transmissions matching the specified series of message classes.

Proof: If this were decidable, then the procedure for deciding it could be applied to the trivial sequence consisting of a single message and thereby used to decide the problem proven undecidable in Theorem V.4.

A Decidable Property of DPMS Models

While Corollary V.3 states that it is not possible, in general, to determine whether a given instantaneous configuration will occur during some computation of an arbitrary DPMS model, it certainly is decidable whether a particular computation is possible for any given DPMS model. This is shown by the folowing:

Theorem V.6: For any given DPMS model M=(P,x<0>,T) and finite-length computation z it can be determined whether z is an element of Z<M>.

Proof: We proceed by induction on the length of z as measured by the number of computation steps

which it contains. Clearly, when z has length one, it is a computation of M if and only if z=x<0>. Now suppose that the theorem is true for any computation of length less than or equal to k for some integer k≥1 and consider a computation z' of length k+1. Then it must be the case that

$$z' = y\langle 0,1\rangle \bullet y\langle 1,2\rangle \bullet \ldots \bullet y\langle k-2,k-1\rangle \bullet y\langle k-1,k\rangle$$

for some appropriate set of purported computation steps. By the induction hypothesis we can decide whether

$$z'' = y\langle 0,1\rangle \bullet y\langle 1,2\rangle \bullet \ldots \bullet y\langle k-2,k-1\rangle$$

is a computation of M. If not, then z' cannot be a computation for M. If z" is an element of Z<M>, then we have only to ascertain whether y<k-1,k> is a possible computation step for M. This can be done by checking the finite number of basic execution cycles possible for M from instantaneous configuration x<k-1> and determining if any of them yields the desired x<k>, i.e., the final instantaneous configuration of z'. If so, then z' is an element of Z<M>, otherwise it is not. Thus, if the theorem holds for computations of length less than or equal to k then it holds for computations of length k+1. Therefore, knowing that the theorem holds for computations of length one, we have shown that it holds for any finite-length computation.

This theorem does not, of course, contradict Theorem V.2 or Corollary V.3. The ability to decide whether a given finite-length computation is possible for a given model is not in general sufficient to permit a determination as to whether a given instantaneous configuration can appear during any of the model's possible computations. Using the technique of Theorem V.6 to produce a definitive answer to that question would, for an arbitrary DPMS model, require the checking of infinitely many possible computations. A potentially infinite undertaking does not qualify as a decision procedure, so Theorem V.2 and Corollary V.3 are not jeopardized by Theorem V.6.

A Final Undecidability Result

The fact that Theorem V.6 can be established suggests that a similar result might be obtainable for message sequences. That is, although we have shown it to be undecidable whether a given message or series of messages will appear in any of the possible message sequences of a given DPMS model, it could still be possible to determine whether a given string of message classes is a viable message sequence for the model, i.e., corresponds to the message transmission sequence arising from some model computation. However, it turns out that this problem too is undecidable. The proof of this fact hinges upon a Dynamic Process Modelling Scheme construction for another universal

computation model, the two counter machine.

Theorem V.6 states, in essence, that for any given DPMS model we can check any finite number of alleged computation steps for legality under the formal semantics of DYMOL. This is possible because each computation step simply consists of two instantaneous configurations and the only possible activity which can occur between those two configurations is a single DPMS basic execution cycle, which is guaranteed to terminate by the formal semantics of DYMOL. Therefore, checking is reduced to the consideration of a small number of cases. A message sequence, however, arises from a message transmission sequence, which is a sequence of SEND and/or RECEIVE instruction executions, each potentially separated from the next by arbitrarily many executions of statements which cause no message transmission activity. Thus, checking can be vastly more difficult for a message sequence than for a computation in any given DPMS model. Indeed, we will show that the computation of an arbitrary Turing machine can be simulated between the executions of a pair of DYMOL message transmission instructions in such a way that no message transmission instructions are executed during the simulation. The undecidability of the halting problem for Turing machines will then enable us to prove that it is undecidable whether a given DPMS model can generate a given message sequence.

The necessity for two different Dynamic Process Modelling Scheme constructions of universal computation models might well be questioned. Indeed, the construction which we are about to exhibit would obviously have been sufficient to establish Theorems V.2 and V.4 and their corollaries as well as the theorem in whose proof it will be used. However, the upcoming construction relies upon some of the properties of set variables, which are distinctly necessary but not the most basic features of DYMOL, while the previous construction was intentionally as conservative as possible in terms of the DPMS features employed in its formulation. Thus, while one could envision a modelling scheme without set variables in which the message sequence question might be decidable, a scheme which lacked any of the features employed in our earlier construction could scarcely be considered suitable to PSDS modelling. We therefore feel much more confident in arguing that the undecidability results established above are indigenous to parallel systems with dynamic structure and in no way result from artifacts of the Dynamic Process Modelling Scheme.

It can be shown [Mins67] that an arbitrary Turing machine can be simulated by a simple device known as a <u>two counter machine</u>. Such a machine consists of two registers which we will denote by b and c, each capable of holding any non-negative integer, and a program composed from two types of instructions. One instruction simply increments one of

the registers by unity and is denoted by either b' or c' depending upon the register involved. The other instruction is denoted by either b⁻(n) or c⁻(n) and indicates that the specified register is to be decremented by one, unless that register currently contains a zero, in which case the register's contents remain unchanged and the program instruction labelled n becomes the next instruction to be executed. This two counter machine, coupled with the ability to preset the contents of the registers b and c, is sufficient for simulating a Turing machine. Hence, a Dynamic Process Modelling Scheme construction capable of simulating a two counter machine can be used to establish the undecidability result we seek.

The construction of a DYMOL process template simulating a two counter machine is reasonably straightforward. The two registers can be represented by set variables, which we will call /b/ and /c/. The non-negative integer contents of the registers can then be represented by the number of elements in the respective multisets /b/ and /c/, with empty multisets corresponding to a zero value for a register's contents. A pair of initial CREATE instructions such as:

CREATE someproc b and CREATE someproc c

serve to obtain non-null values, namely process identifiers for some otherwise unused processes, for use in this unary representation of integers in /b/ and /c/. The registers' initial values may either be set in the initial

instantaneous configuration of the model containing the simulated two counter machine or by a preliminary series of incrementing steps.

Using the representation of registers and numbers outlined above, the instructions b' and c' may then be represented by the DYMOL assignments:

tci: /b/ := b and tcj: /c/ := c

respectively. Each assignment serves to add one element to the specified multiset, thus increasing by one the value represented by its contents. The conditional decrement and branch instruction b⁻(n) takes the following form in DYMOL:

```
tck: CREATE someproc boolk;
tcm: FOR SOME d :- /b/ DO
tcn:  DESTROY boolk
```

and c⁻(n) is identical except for the replacement of /b/ by /c/. Each such decrement instruction employs a separate variable in the role of 'boolk' whose contents name the process identifier whose presence or absence in the active process set A connotes the outcome of the conditional decrement. According to the semantics of the DYMOL partial selection construct, the destructive assignment and the DESTROY instruction will only be executed when the set variable is non-empty. Thus, only in the case that the set variable is non-empty, i.e., represents a non-zero register content, will one element be removed from that set variable and the designated process destroyed. This effectively

results in a decrementing of the simulated register's contents. When the set variable is already empty at the time statement tcm is executed, no simulated decrementing occurs and the process designated by variable boolk is not destroyed. Hence a subsequent conditional construct of the form:

```
tcp: IF boolk IN A THEN ... ELSE ...
```

or an indefinite iteration (WHILE) using the same type of condition, can be used to determine whether the register in question contained a zero at the time of the decrement instruction's execution and, therefore, whether a branch to the instruction labelled n should occur.

Since DYMOL has no goto instruction, the conditional branching aspect of the two counter machine's decrement instruction cannot be modelled directly. However, DYMOL does have a conditional (IF...THEN...ELSE...) construct, an iteration (WHILE) construct and a sequencing construct (the BEGIN...END block). Moreover, through the use of CREATE, DESTROY and ...IN A in the manner illustrated above, an arbitrary number of boolean conditions can be represented in any DYMOL process template. Therefore, the famous result of Bohm and Jacopini [Bohm66] guarantees that any control flow specified using goto statements can be represented using the DYMOL constructs. This fulfills the final function required to permit the construction of any two counter machine within a DYMOL process template, without using any SEND or RECEIVE

instructions. Using the construction technique outlined above, we can now establish the undecidability result discussed earlier in this section.

Theorem V.7: It is undecidable whether, for a given DPMS model M=(P,x<0>,T), a given sequence of messages is a message sequence which could result from some computation of M.

Proof: Suppose it were decidable. Then the halting problem would be decidable for any arbitrary Turing machine as follows: (1) Produce a specification of the Turing machine as a two counter machine. (2) Generate a DPMS model consisting of a single process which a) sends some message m1, then b) simulates the two counter machine produced at step (1) in the manner outlined in this section, and c) sends some message m2 upon completion of that simulation. (3) Determine whether the message sequence 'm1 m2' results from any computation of this model. The Turing machine in question halts if and only if 'm1 m2' is a possible message sequence for the DPMS model so constructed. But since the halting problem for an arbitrary Turing machine is undecidable, this results in a contradiction and hence it must be undecidable whether a given sequence of messages is a possible message

sequence for an arbitrary DPMS model.

Significance of the Results

In this chapter we have established several undecidability results related to DPMS models of parallel systems with dynamic structure. While these results are interesting and have ramifications for the use of the modelling scheme, their significance to the usefulness of the Dynamic Process Modelling Scheme should not be overestimated. Although the theorems and corollaries rule out some approaches to the analysis of DPMS-modelled parallel systems with dynamic structure, they should not be construed as demonstrating that the scheme could not be useful in such applications as the design and analysis of dynamically-structured, concurrent software systems.

The results of this chapter do indicate that there is no hope for discovering an algorithm capable of deciding whether a given instantaneous configuration can be achieved by an arbitrary DPMS model. Similarly, they show that no effective procedure can be found for determining the message behavior of an arbitrary PSDS modelled in the Dynamic Process Modelling Scheme. These facts suggest that any automated system using DPMS as a basis for software design and analysis should rely upon techniques such as 'feedback analysis' [Ridd77b] or perhaps heuristic-based analysis

methods rather than attempting to formally discover the behavioral characteristics of an arbitrary DPMS model. The results also demonstrate that the extended DYMOL discussed in Chapter III, while it may add to the descriptive capabilities of DPMS, does not in fact represent a more powerful computational model than the version which we have employed in this chapter to simulate universal computation models.

The theorems and corollaries do not, however, rule out the possibility that behavioral properties can be proven with respect to a given DPMS model. Indeed, we have given just such a proof in the example discussed in Chapter IV, where we showed that no element of a certain model's terminal instantaneous configuration set could ever appear in any of that model's computations. Nor do the present results mean that these behavioral questions do not admit of decision procedures applicable to whole subclasses of parallel systems with dynamic structure. In fact, the results encourage the search for interesting and appropriate PSDS subclasses and the behavioral analysis methods suitable to them. The next chapter represents a preliminary effort in that direction.

Thus, just as the undecidability of the ambiguity question for context free grammars does not, in practice, severely limit the usefulness of these grammars, we do not

anticipate that the undecidability results of this chapter will nullify the utility of the Dynamic Process Modelling Scheme. While such results dissuade us from seeking algorithms with universal DPMS applicability, it remains reasonable to seek interesting decidable subclasses and to use the rigorous formal structure provided by DPMS in the construction of proofs of particular properties for particular PSDS instances. The example of Chapter IV and the subclass and techniques discussed in the next chapter represent initial efforts along these lines.

CHAPTER VI

MODELLING PARALLEL SYSTEMS WITH DYNAMIC CONNECTIVITY

The Dynamic Process Modelling Scheme provides the capability for modelling a very wide range of parallel systems with dynamic structure. However, the results of Chapter V have shown that many interesting behavioral questions are undecidable for models constructed using the full power of DPMS. Therefore, in this chapter we consider a subclass of parallel systems with dynamic structure, its DPMS description and a technique for deriving a behavioral representation, called a constrained expression, for models of systems in this subclass. While other PSDS subclasses also bear investigation, the one discussed in this chapter corresponds to a reasonably common and interesting software system organization and has been selected for study here on that basis.

Dynamic Connectivity

One interesting subcase of parallel systems with dynamic structure concerns those systems having a fixed set of components but a varying pattern of intercomponent communication paths. Examples of systems exhibiting this type of structure include computer networks and operating systems supporting intertask message transfer for a fixed set of tasks. We refer to this PSDS subclass as parallel systems with dynamic connectivity, and frequently abbreviate the term 'dynamic connectivity' by 'DC'.

Representation of this subclass of systems in the Dynamic Process Modelling Scheme requires only a subset of the scheme's features. In particular, the process templates of such systems may be described using just a subset of the Dynamic Modelling Language, and the range of possible alterations to instantaneous configurations brought about by computation steps is reduced in DPMS models of DC systems.

The DYMOL subset used in describing process templates in a DPMS model of a parallel system with dynamic connectivity is called DC DYMOL. It differs from the basic version of DYMOL discussed in Chapters III and IV in three significant ways. First, DC DYMOL lacks the DYMOL process structure instructions, i.e., CREATE and DESTROY, which serve no function when the process structure of the modelled

system is fixed. Secondly, DC DYMOL has no variables or assignment instructions other than buffer assignment, since these features are useful primarily in modelling the manipulation of a system's process structure. Finally, the selection constructs of DYMOL, i.e., FOR SOME and FOR ALL, are not found in DC DYMOL, since they require variables and assignments in their specification. The complete BNF description of DC DYMOL may be found in Appendix D, while the formal semantics of its instructions are simply the appropriate subset of those described in Chapter IV.

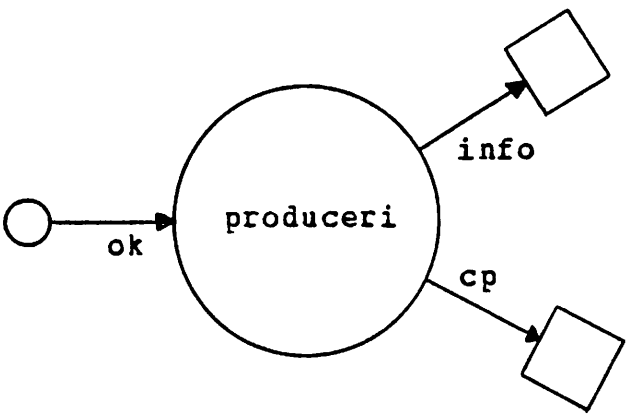
Instantaneous configurations for DPMS models of parallel systems with dynamic connectivity are of exactly the same form as those described in Chapter IV. However, the limitations imposed by the DC context have several implications for the instantaneous configurations of such models. Their process states, for instance, are simpler than those found in instantaneous configurations of the full modelling scheme since they never contain any of the v(i) or f(i) multisets used in representing variables and total selection constructs. Also, due to the fixed set of processes in a DC model, the active process set A is constant for all instantaneous configurations of any model computation. Lastly, the absence of CREATE and DESTROY implies that the configuration matrix C is a fixed-size, square matrix having a constant set of indices throughout any computation of a given DC model. The contents of C's

entries may, of course, change during a model's computations, but the DC limitations even enable us to establish upper bounds on the cardinality of each matrix entry, i.e., on the number of elements in each set of port name ordered pairs in C.

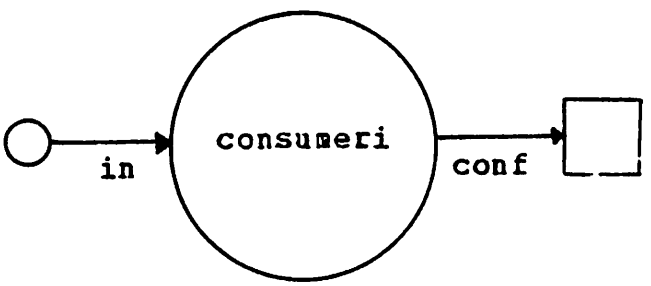
An example of a DPMS model of a parallel system with dynamic connectivity appears in Figures VI-1, VI-2 and VI-3. The example describes a producer-consumer situation in which a producer generates information which can be processed by either of two consumers. The producer in the modelled system generates a stream of variable-length information packets, each packet postfixed with a termination indicator. It is intended that each packet constitute a single complete, coherent set of data for a consumer. Therefore, proper processing requires that each complete packet, including its termination indicator, be received by exactly one consumer. Each time that the producer is prepared to generate an information packet, a manager process called c_pool selects a consumer to receive the information. When the producer has completed the generation of a packet, c_pool is notified to disconnect the producer from the most recently active consumer.

The process templates of Figure VI-1 give the DC DYMOL descriptions of the producer and consumer processes of this model. Following each sending of the 'ready' message

```
producer: p1: WHILE INTERNAL TEST DO
              BEGIN
          p2:   SET BUFFER := ready;
          p3:   SEND cp;
          p4:   RECEIVE ok;
          p5:   WHILE INTERNAL TEST DO
                  BEGIN
          p6:       SET BUFFER := goods;
          p7:       SEND info;
          p8:       RECEIVE ok
                  END;
          p9:   SET BUFFER := term;
         p10:   SEND info;
         p11:   SET BUFFER := done;
         p12:   SEND cp
              END.
```




```
consumer: c1: DO FOREVER
              BEGIN
          c2:   RECEIVE in;
          c3:   IF BUFFER = goods THEN
                  BEGIN
          c4:       SET BUFFER := got_it;
          c5:       SEND conf
                  END
              END.
```



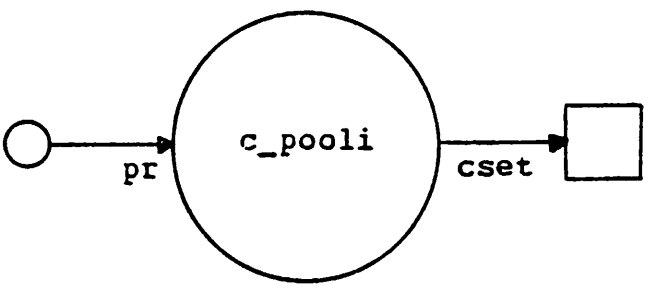
Producer and Consumer Process Templates

Fig. VI-1

indicating its intention to generate an information packet, the producer (at p4) awaits an indication that a consumer is prepared to receive it before actually commencing generation of the information. The consumer's confirmation of each reception of a 'goods' message (c3, c4 and c5) is used by the producer (at p8) to ensure that the items in an information packet are consumed in the same order in which they were produced. The producer's 'term' message (p9 and p10) is the packet termination indicator sent to the consumer, while the 'done' message (p11 and p12) informs the consumer pool manager, c_pool, that a complete packet has been generated.

The process template describing the manager of the pool of available consumer processes, called c_pool, is shown in Figure VI-2. When informed that the producer is ready to generate an information packet this process selects a consumer to handle the packet based upon some internal computation which is modelled as a nondeterministic choice (cp3). The selected consumer is placed in communication with the producer (cp4 and cp5 or cp10 and cp11) and then the manager so informs the producer (cp6 or cp12). When informed that a complete packet has been generated (cp7 or cp13), the manager dissolves the communication linkage between the two processes (cp8 and cp9 or cp14 and cp15), and awaits notification (at cp2) that another information packet is about to be generated. It is easily seen that

```
c_pool: cp1: DO FOREVER
               BEGIN
        cp2:     RECEIVE pr;
        cp3:     IF INTERNAL TEST THEN
                   BEGIN
        cp4:         ESTABLISH producer1.info consumer1.in;
        cp5:         ESTABLISH consumer1.conf producer1.ok;
        cp6:         SEND cset;
        cp7:         RECEIVE pr;
        cp8:         CLOSE producer1.info consumer1.in;
        cp9:         CLOSE consumer1.conf producer1.ok
                   END
                 ELSE
                   BEGIN
       cp10:         ESTABLISH producer1.info consumer2.in;
       cp11:         ESTABLISH consumer2.conf producer1.ok;
       cp12:         SEND cset;
       cp13:         RECEIVE pr;
       cp14:         CLOSE producer1.info consumer2.in;
       cp15:         CLOSE consumer2.conf producer1.ok
                   END
                END.
```



Consumer Pool Manager Process Template

Fig. VI-2

both this template and those of Figure VI-1 use only the constructs and instructions of DC DYMOL, and therefore that this model falls within the dynamic connectivity subclass of parallel systems with dynamic structure.

The complete model is shown, formally and graphically, in Figure VI-3. The simplified process states of Q<0> are typical of DPMS models of DC systems. The active process set A will be identical to A<0> throughout any model computation, and the size and indices of C will similarly be unchanged during the modelled system's operation. Although the techniques illustrated at the end of Chapter IV could also be used here to study the potential behavior of the example DC system, we will forego such behavioral analysis for the present, returning to it after we develop a new tool, constrained expressions, for use in that analysis.

Constrained Expressions

In conjunction with his program process modelling scheme, Riddle developed a behavioral representation for models of statically-structured concurrent software systems called message transfer expressions, or MTEs [Ridd72]. This representation technique, which focuses on message transmission activity within the modelled system, was later generalized to encompass behavioral descriptions in terms of any specified set of significant events rather than simply

M = (P, x<0>)

where

P = {producer, consumer, c_pool}

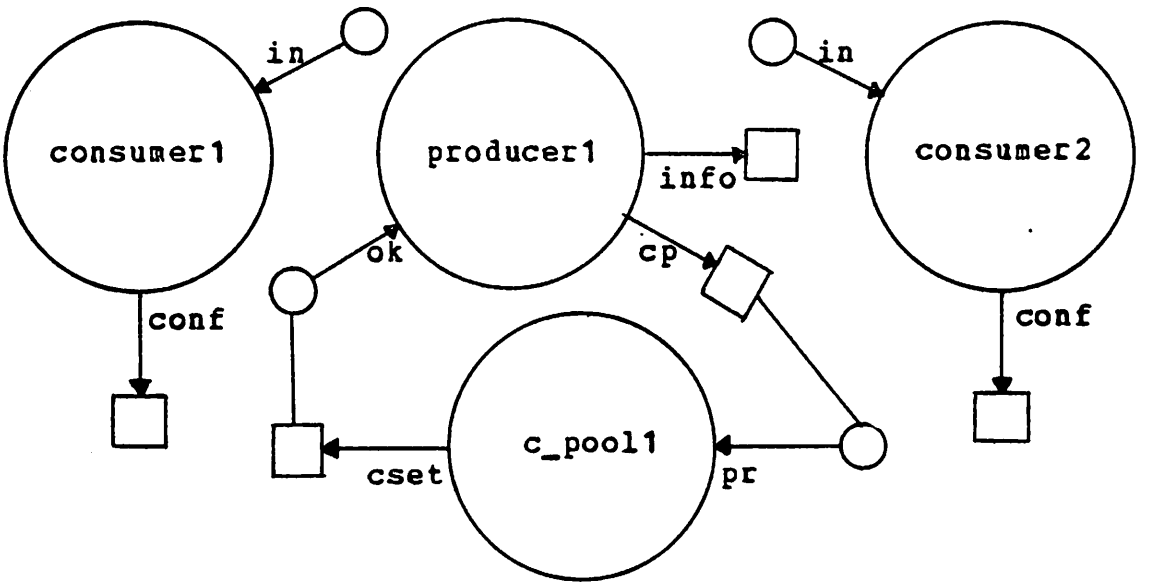
x<0> = (C<0>, A<0>, Q<0>, L<0>)

C<0> =

	producer1	consumer1	consumer2	c_pool1
producer1	∅	∅	∅	(cp,pr)
consumer1	∅	∅	∅	∅
consumer2	∅	∅	∅	∅
c_pool1	(cset,ok)	∅	∅	∅

A<0> = {producer1, consumer1, consumer2, c_pool1}

Q<0> = ((p1,∅),(c1,∅),(c1,∅),(cp1,∅)) L<0> = ((∅,∅),∅,∅,∅)



The Example DC System Model

Fig. VI-3

message transfer events. This generalized behavioral representation scheme is known as event expressions [Ridd76]. It has been shown in both [Ridd72] and [Ridd76] that effective procedures exist for deriving a message transfer expression from a model in the program process modelling scheme such that the resulting MTE represents all of the possible message transmission behaviors of the modelled system. Welter has defined an alternative formulation of message transfer expressions, called counter expressions [Welt76], and demonstrated an effective procedure for deriving counter expressions from models described using a variant of the program process modelling scheme. Constrained expressions, which we develop in the sequel, are a generalization of counter expressions applicable to models of dynamically-structured parallel systems. We present a general definition of constrained expressions in this section, then in the next section discuss a particular version related to parallel systems with dynamic connectivity.

A complex software system's activity may be viewed as consisting of events, where the events can be of arbitrarily long duration and arbitrary complexity, depending upon the level of detail at which the system is being considered. For instance, viewed at one level a system's events might involve bit manipulations in registers or memory while from another viewpoint the same system's events might be entire

procedure executions or the movement of messages into and out of links. Each possible behavior for a system can thus be described as a sequence of events. Such a sequence constitutes a string over an alphabet of event names defined for the given system and given viewpoint level. Since a complex software system can have a very large number of distinct possible behaviors, it is useful to define a finite representation for the set of event sequences corresponding to those behaviors. Such a set of strings constitutes a language over the event alphabet. In the theory of formal languages and automata, regular expressions can be used to define those sets of strings over specified alphabets which are known as regular languages. In an analogous fashion, event expressions, counter expressions and constrained expressions all define certain languages over event alphabets such that the strings of those languages represent the possible behaviors of complex software systems.

Riddle's event expressions define languages over an alphabet, E, of distinguished events. The expressions are composed of symbols from E, symbols from an auxiliary alphabet S of synchronization symbols, the special symbols λ and $\emptyset$, and a set of operator symbols. The event expression operators include the familiar operators of regular expressions -- union (represented by $\cup$), concatenation (represented by juxtaposition), and transitive closure (represented by *) -- plus two operators, $\sqcap$ and $^+$, used to

represent concurrent activity. All the event expression operators are associative and their relative precedence is that shown in Figure VI-4. The function of the first three operators is identically that which they connote in regular expressions. The ⧢ operator signifies the shuffling or interleaving of the two strings which are its operands[1], while the unary operator + denotes the interleaving of zero or more copies of its operand. The examples of Figure VI-4 illustrate the meaning of these various event expression operators; a more complete and formal discussion may be found in [Ridd76].

As described above, event expressions contain synchronization symbols in addition to symbols from E and the special symbols λ (null event sequence) and $\emptyset$ (empty set of sequences). Thus, following interpretation of the event expression operators, an event expression initially yields a set of strings over the augmented alphabet $(E \sqcup S)$. In order to determine the set of event strings, i.e., the language over E, represented by an event expression, a cancellation rule is applied to the synchronization symbols in the strings of the initial set. Following application of this rule, any strings in which one or more synchronization symbols remain are discarded, leaving a set of strings containing only symbols from E. This remaining set is the

[1]The shuffle operation was first defined and studied by Ginsburg [Gins66] in the context of formal languages.

Operator Precedence -- highest to lowest

1. * and +
2. juxtaposition
3. ▫
4. ␣

Event Expression Operator Examples

$(x\ y\ ␣\ z\ w)^* = \{x\ y,\ z\ w,\ x\ y\ x\ y,\ z\ w\ z\ w,\ x\ y\ z\ w,\ z\ w\ x\ y,\ x\ y\ x\ y\ x\ y,\ z\ w\ z\ w\ z\ w,\ \ldots.\}$

$x\ y\ ▫\ z\ w = \{x\ y\ z\ w,\ x\ z\ y\ w,\ x\ z\ w\ y,\ z\ x\ y\ w,\ z\ x\ w\ y,\ z\ w\ x\ y\}$

$(x\ y)^+ = \lambda\ ␣\ x\ y\ ␣\ x\ y\ ▫\ x\ y\ ␣\ x\ y\ ▫\ x\ y\ ▫\ x\ y\ ␣\ \ldots$
$= \{\lambda,\ x\ y,\ x\ y\ x\ y,\ x\ x\ y\ y,\ x\ y\ x\ y\ x\ y,\ \ldots\}$

Cancellation Rule Example

If $E = \{x, y\}$ and $S = \{@, @'\}$ where $@$ and $@'$ are assumed to cancel:

$x\ @\ y\ ▫\ z\ @'\ w = \{x\ @\ y\ z\ @'\ w,\ x\ @\ z\ y\ @'\ w,\ x\ z\ @\ y\ @'\ w,\ z\ x\ @\ y\ @'\ w,\ x\ @\ z\ @'\ y\ w,\ x\ z\ @\ @'\ y\ w,\ z\ x\ @\ @'\ y\ w,\ x\ @\ z\ @'\ w\ y,\ x\ z\ @\ @'\ w\ y,\ z\ x\ @\ @'\ w\ y,\ x\ z\ @'\ @\ y\ w,\ z\ x\ @'\ @\ y\ w,\ z\ @'\ x\ @\ y\ w,\ x\ z\ @'\ @\ w\ y,\ z\ x\ @'\ @\ w\ y,\ z\ @'\ x\ @\ w\ y,\ x\ z\ @'\ w\ @\ y,\ z\ x\ @'\ w\ @\ y,\ z\ @'\ x\ w\ @\ y,\ z\ @'\ w\ x\ @\ y\}$

Cancellation deletes all but the sixth, seventh, ninth, tenth, eleventh, twelfth, fourteenth and fifteenth elements of the set, leaving:

$x\ @\ y\ ▫\ z\ @'\ w = \{x\ z\ y\ w,\ z\ x\ y\ w,\ x\ z\ w\ y,\ z\ x\ w\ y\}$

Event Expression Operator Properties

Fig. VI-4

language over E represented by the event expression. The synchronization symbol cancellation rule is intended to enforce the ordering of events which necessarily precede or follow one another in the complex software system behaviors represented by a given event expression. In essence, the rule permits the cancellation of designated pairs of synchronization symbols when they occur immediately adjacent to each other in a string of symbols over (E ⊔ S). A simple example of the application of the cancellation rule for event expressions appears in Figure VI-4.

When the set of event symbols, E, is limited to the set of message types defined in a program process model and the distinguished events are taken to be the transfer of messages into and out of links, Riddle's event expressions are known as message transfer expressions. Two different effective procedures have been described for deriving the MTE representing all of a modelled system's possible behaviors (in terms of message transmissions) from a model constructed using the program process modelling scheme. These techniques are described in [Ridd72] and [Ridd76], while a similar algorithm for deriving parameterized message transfer expressions useful in software system performance prediction is presented by Sanguinetti in [Sang77].

Welter's counter expressions provide an alternative formulation for message transfer expressions. For our

purposes, the significant advantages in Welter's approach are its simpler derivation procedure and more transparent interpretation properties as compared to the earlier MTE presentations. Counter expressions, like event expressions, are defined relative to a pair of alphabets -- a terminal alphabet E and an auxiliary alphabet S. The counter expression itself is a regular expression, formed over the augmented alphabet (E ⊔ S) through the use of any of the event expression operators other than +.[1] Such an expression is said to generate an interpreted language over E, which can be derived by intersecting the regular set described by the counter expression with a constraining language C, and then applying a mapping which takes the result of that intersection into E*. A formal definition of this procedure, which is very similar to that given below for constrained expressions, appears in [Welt76]. Informally, the procedure may be described as checking each string defined by the counter expression to determine if its sequence of symbols from S corresponds to the specified constraint, discarding the string if it does not and otherwise retaining the string with all its symbols from S erased. The constraint used in the comparison is itself a language over S, described by an expression over S using any of the event expression operators, including +. Welter has

[1]Proofs that such expressions, also called shuffle expressions in [Ridd76], describe regular languages may be found in [Gins66], [Ridd76], and [Welt76].

shown how counter expressions can be derived from models of complex concurrent software systems described using a variant of Riddle's program process modelling scheme. It has also been shown that, when interpreted using the constraining language specified in [Welt76], such a derived counter expression represents precisely the set of all possible message transmission behaviors of the modelled system.

Our constrained expressions are simply a generalization of counter expressions allowing for the application of multiple constraints in determining the acceptability of a string. As with the previously-defined expressions, we begin with a representation over an augmented alphabet and use an interpretation rule to produce a set of strings over the actual alphabet of interest. In particular, we let E and S be two disjoint, finite sets called the event alphabet and the constraint alphabet, respectively. A constraint set, CS, consisting of n constraining languages, Ci for $1 \leq i \leq n$, can then be defined with respect to n disjoint subsets, Si, of S. Each such Ci is represented by an expression over Si, ex(Si), formed using any of the event expression operators, and interleaved with E* and Sj* for $j \neq i$. That is,

$Ci = ex(Si) \square S1^* \square \ldots \square Si-1^* \square Si+1^* \square \ldots \square Sn^* \square E^*$

for each constraining language Ci, $1 \leq i \leq n$. A constrained expression with respect to CS is then defined to be any

expression over (E ⊔ S) which can be formed using the event expression operators other than +. This expression thus represents a regular language, L', which is a subset of (E ⊔ S)* and which we call the uninterpreted language of the constrained expression. We also define a homomorphism H:(E ⊔ S)* -> E* by:

$$H(e) = e \qquad \text{for all } e \text{ in } E$$

$$H(s) = \lambda \qquad \text{for all } s \text{ in } S$$

Finally, for a given constrained expression with respect to CS which represents the uninterpreted language L', we define the interpreted language, L, represented by the expression to be the set of strings over E (i.e., subset of E*) described by:

$$L = H(L' \sqcap C1 \sqcap \ldots \sqcap Cn) \qquad \text{for } Ci \text{ in } CS$$

Except for some notational changes, this definition is based directly upon that given for counter expressions in [Welt76]. Indeed, when n=1, S={@i,@i' | 1≤i≤m} for some positive integer m, and CS={C1} where

$$C1 = (@1\ @1')^+ \sqcap \ldots \sqcap (@m\ @m')^+ \sqcap E^*$$

constrained expressions with respect to CS are nothing other than counter expressions.

Rather than construct an artificial example to illustrate these general constrained expression definitions we will proceed directly to consideration of a particular instance, namely the DC constrained expression. The example presented in conjunction with that discussion should amply

illuminate the various aspects of constrained expressions and their definitions.

DC Constrained Expressions

Having discussed both parallel systems with dynamic connectivity and constrained expressions in this chapter, we now link the two by considering the particular class of constrained expressions which is relevant to DC systems. In this section we define DC constrained expressions and in the next section we present an algorithm for their derivation from DPMS models of parallel systems with dynamic connectivity. In the section following that we offer an example illustrating the algorithm, the expressions and the use of both in analyzing models of DC systems. The chapter concludes with a section in which we show that a DC constrained expression derived from a DPMS model according to our algorithm precisely represents the possible message transfer behaviors of the modelled system.

DC constrained expressions are intended to represent the possible message transmission behaviors of DPMS models of parallel systems with dynamic connectivity. In particular, each movement of a message of a given type into a link (via a SEND) or out of a link (via a RECEIVE) is viewed as an event symbolized by the appropriate message class identifier. The source and destination of such

message movements are not explicitly represented in this description, although some information along these lines can be obtained by a DPMS modeller through utilization of sufficiently numerous and varied message classes. Thus the event alphabet Σ for the DC constrained expression corresponding to a given DPMS model of a DC system is just the set of all message class identifiers used in that model.

As will be seen below, the actual constrained expression for a DPMS DC model consists of the interleave of a set of subexpressions, with one subexpression corresponding to the sequential activity of each instantiated process of the modelled system. We distinguish three types of constraints which must be imposed upon the ordering of the symbols in strings represented by these interleaved expressions and hence there are three disjoint subsets in the constraint alphabet S for DC constrained expressions.

The first type of constraint relates to the transmission of messages. In any legal DPMS computation, each reception of a message of a given class from some link must be preceded at some point in the computation by a corresponding placement of a message of that type into that link. Hence, a constraint is needed to assure that at any point in a particular behavioral trace at least as many messages of type x have been placed in link y as there have

been messages of type x received from link y. The constraint alphabet subset

$$S1 = \{@i, @i' \mid 1 \leq i \leq mp\}$$

is used in stating this behavioral requirement, where mp is equal to the product of the number of message types and the number of outbound ports (i.e., links) in the modelled system. The corresponding constraining language C1 is represented by the expression[1]:

$$C1 = \left[\!\!\left[\;\right]\!\!\right]_{i} (@i^* \,\square\, (@i\ @i')^+) \,\square\, S2^* \,\square\, S3^* \,\square\, E^*$$

$$= @1^* \,\square\, (@1\ @1')^+ \,\square\, \ldots \,\square\, @mp^* \,\square\, (@mp\ @mp')^+ \,\square\, S2^* \,\square\, S3^* \,\square\, E^*$$

The event expression (b c)+ represents a set which may be described as "all strings containing equal numbers of b's and c's such that any prefix of any string always contains at least as many b's as c's". Thus, if each @i symbol is associated with the placing of a message of type x into link y while each @i' is associated with the receipt of a message of type x from link y, this constraining language describes the situation where the reception of a message is always preceded by a corresponding placement, although more messages may be placed than are ever received during system operation (allowed by the interleaved @i*). This, of course, is exactly the intended constraint upon message

[1]The first part of the expression is an interleave over all values of an index variable, i, for 1≤i≤mp. Our notation here is similar to that commonly used to represent a union or summation over all values of an index variable.

transmission in behaviors of the modelled system.

A second type of constraint on DC system behavior involves the use of interprocess communication channels. According to the semantics of DPMS modelling, a message can only be received through inbound port y from link (i.e., port) x if a communication channel connects x and y at the time of that reception. The constraint alphabet subset

$$S2 = \{\phi i,\ \phi i',\ \varnothing i,\ \varnothing i' \mid 1 \leq i \leq np\}$$

is used in stating this behavioral constraint, where np is equal to the product of the number of inbound ports and number of outbound ports (i.e., links) in the modelled system. The corresponding constraining language C2 is represented by the expression:

$$C2 = \left[\!\!\left[\right]\!\!\right]_i (\phi i\ (\phi i^* \sqcap (\varnothing i\ \varnothing i')^*)\ \phi i'^*)^* \sqcap S1^* \sqcap S3^* \sqcap E^*$$

This constraint is most easily understood by associating the symbol ϕi with the establishment of a channel, $\phi i'$ with the closing of that channel, and the concatenation $(\varnothing i\ \varnothing i')$ with the reception of a message across that channel. The constraint may then be seen to require that a channel be established, either in the model's initial instantaneous configuration or by an ESTABLISH command execution, before a message reception may utilize the channel, as symbolized by the ϕi preceding $(\varnothing i\ \varnothing i')^*$ in the expression. C2 also allows for the occurrence of additional, ineffectual

ESTABLISH command executions once a channel has been established, symbolized by the ϕi* term of the expression. Finally, the constraining language requires that the closing of a channel (i.e., one or more appearances of the symbol ϕi') be separated from any subsequent message receptions using that channel by one or more ESTABLISH command executions. It should be noted that this constraining language does not require the closing of a channel, nor the utilization of an established channel, as a condition for the acceptance of a constrained expression behavior representation. Rather, C2 reflects precisely the constraints upon interprocess communication channel utilization which are levied by the semantics of DPMS modelling.

The final type of constraint on DC system behavior relates to the usage and modification of buffer contents during a given process' operation. In order to accurately represent the dependency of message sending and some iterative and conditional statement executions on the contents of the process' buffer, we use a third subset of the constraining alphabet

$$S3 = \{\#i,j,\ \#i,j' \mid 1 \leq i \leq k,\ 1 \leq j \leq m+1 \}$$

where k is the number of instantiated processes and m the

number of distinct message types in the modelled system.[1] The extra second subscript value allows for representing an empty buffer. The constraining language corresponding to S3 is:

$$C3 = \left[\begin{array}{c}\ \\ \ \end{array}\right]_{i} \left(\left[\begin{array}{c}\ \\ \ \end{array}\right]_{j} (\#i,j\ \#i,j'^{*})^{*} \right) \square S1^{*} \square S2^{*} \square E^{*}$$

Again, interpretation of C3 is facilitated by associating the symbol #x,y with the placing of message class y into the buffer of process x and the symbol #x,y' with a usage of the contents of the buffer of process x at a time when it contains message class y. Such a usage might be part of the evaluation of a condition in an iterative or conditional instruction or it might be a SEND instruction execution. In any case, the constraint C3 is easily seen to require that, for each modelled process, any usages of the process' buffer which presume that it contains a given message class must be preceded by placement of that message class into the buffer, with no intervening modifications to the buffer. The constraint does not, however, require that any usages actually occur between successive modifications of the buffer's contents, due to the #i,j'* term in the expression. Thus C3, like C1 and C2, accurately captures the constraint imposed by DPMS modelling semantics.

[1]The double subscripting of symbols in S3 is used here purely for notational convenience. Obviously only brevity in the representation of C3 would be lost by an appropriate substituting of single subscripts for the double subscripts used here.

C3 also represents the final constraining language for DC constrained expressions, hence the constraint alphabet S for DC constrained expressions is:

$$S = S1 \sqcup S2 \sqcup S3$$

and the constraint set for DC constrained expressions is:

$$CS = \{ C1, C2, C3\}$$

Based upon the definitions of E, S and CS given in this section, we say that a DC constrained expression is a constrained expression with respect to CS which is formed using the augmented alphabet $E \sqcup S$, and proceed to give an algorithm for deriving a DC constrained expression from a DPMS model of a DC system.

The Derivation of DC Constrained Expressions

Suppose that we are given a DPMS model, M[DC], of a parallel system with dynamic connectivity having k instantiated processes, m distinct message types, n inbound ports and p outbound ports. The DC constrained expression corresponding to this model may then be derived as follows:

We first define four mappings which assign unique integer identifiers to processes, message types, inbound ports and outbound ports, i.e.,

K: {M[DC]'s process identifiers} -> {1,...,k}
M: {M[DC]'s message types} -> {1,...,m}
N: {M[DC]'s inbound port names} -> {1,...,n}
P: {M[DC]'s outbound port names} -> {1,...,p}

Three more mappings are also needed:

$$MP: \{1,\ldots,m\} \times \{1,\ldots,p\} \rightarrow \{1,\ldots,mp\}$$
$$PN: \{1,\ldots,p\} \times \{1,\ldots,n\} \rightarrow \{1,\ldots,pn\}$$
$$KM: \{1,\ldots,k\} \times \{1,\ldots,m+1\} \rightarrow \{1,\ldots,k(m+1)\}$$

These mappings allow us to assign subscripts to the various symbols of the contraint alphabet which are generated during the construction of the DC constrained expression.

The constrained expression which we will derive consists of a subexpression INIT followed by (i.e., concatenated with) k interleaved subexpressions PEXPi, $1 \le i \le k$. That is, the derived DC constrained expression has the form:

$$\text{INIT } (\underset{i}{\square} \text{ PEXPi})$$

The INIT subexpression represents the status of the links, buffers and interprocess connectivity of the modelled system as reflected in its initial instantaneous configuration and is composed entirely of symbols from the constraint alphabet S. Each of the k PEXPi subexpressions corresponds to one of the k instantiated processes of M[DC] and is derived from the appropriate DC DYMOL process template through the statement-by-statement application of a set of derivation rules.

The INIT subexpression is simply the concatenation of a series of symbols from S, determined by inspection of the initial instantaneous configuration of M[DC]. Symbols from

S1 are added to the INIT subexpression according to the initial complete link state L<0> of M[DC] as follows: for each message of type x which is in link y as indicated by L<0>, the symbol @i where i=MP(M(x),P(y)) is included in the INIT concatenation. Symbols from S2 are added to the INIT subexpression according to the contents of the initial configuration matrix, C<0>, of M[DC] as follows: for each pair of port names z.x and w.y such that the ordered pair (x,y) is an element of C<0>(z,w), the symbol @i where i=PN(P(z.x),N(w.y)) is included in the INIT concatenation. Finally, symbols from S3 are added to the INIT subexpression according to the buffer contents (bu) components of the process states in Q<0> of M[DC]. For each process state q[x] in Q<0>, if bu[x]=y≠∅ then the symbol #i where i=KM(K(x),M(y)) is included in the INIT concatenation, while if bu[x]=∅ then the symbol #j where j=KM(K(x),m+1) is included in the INIT concatenation. Thus the derived DC constrained expression begins with a series of constraint alphabet symbols which encode the initial status of links, buffers and interprocess communication pathways in the model M[DC].

The k subexpressions PEXPi, 1≤i≤k, whose interleave constitutes the remainder of the derived DC constraint expression are formed through a statement-by-statement application of the rules listed in Figure VI-5. Each rule indicates that a DC DYMOL statement of the given form is to

SEND x => (| | #KM(K(pid),M(y)) y @MP(M(y),P(pid.x)))
y ⊔ #KM(K(pid),m+1))'

RECEIVE x =>
(| | (| | @PN(P(z),N(pid.x)) @MP(M(y),P(z))' y
y z @PN(P(z),N(pid.x))' #KM(K(pid),M(y))))

ESTABLISH x y => @PN(P(x),N(y))

CLOSE x y => @PN(P(x),N(y))'

SET BUFFER := x => #KM(K(pid),M(x))

DO FOREVER <statement> => (<statement>)*

WHILE INTERNAL TEST DO <statement> => (<statement>)*

WHILE BUFFER = x DO <statement> =>
(#KM(K(pid),M(x)) <statement>)*

IF INTERNAL TEST THEN <simple statement> ELSE <statement> =>
(<simple statement> ⊔ <statement>)

IF INTERNAL TEST THEN <statement> => (<statement> ⊔ λ)

IF BUFFER = x THEN <simple statement> ELSE <statement> =>
((#KM(K(pid),M(x)) <simple statement>) ⊔ <statement>)

IF BUFFER = x THEN <statement> =>
((#KM(K(pid),M(x)) <statement>) ⊔ λ)

<statement list>; <statement> =>
(<statement list> <statement>)

DC Constrained Expression Derivation Rules as Applied to the DC DYMOL Process Template of Process 'pid'

Fig. VI-5

be transformed into the symbol sequence signified to the right of the '=>' symbol. For each instantiated process with process identifier pid, the rules are applied to the process template corresponding to that process, making the appropriate substitutions for self-references (implicit and explicit) in the DYMOL program. Thus, for example, where the first rule specifies a statement of the form 'SEND x' (the SEND instruction involves an implicit self-reference since processes may only send and receive through their own ports), its application to the template of process pid is as though the statement actually read 'SEND pid.x'. We therefore find pid.x as the operand of the P mapping in this rule. Similarly, any appearance of the DYMOL reserved word ME (an explicit DYMOL self-reference) in the template is replaced by 'pid' prior to application of the appropriate rule.

The application of the rules to DC DYMOL's basic statements, i.e., the first five rules of Figure VI-5, is a simple matter of replacing the statement by the indicated sequence of symbols from $\Sigma \sqcup S$. The meaning of the remaining rules, i.e., those which apply to the control constructs of DC DYMOL, may be less obvious. In all cases these rules merely require that the indicated action be performed on the expression resulting from application of the appropriate rule to the embedded statement(s) of the control construct. Thus, for instance, the statement:

WHILE INTERNAL TEST DO ESTABLISH proc1.out proc2.in would be transformed into:

(@PN(P(proc1.out),N(proc2.in)))*

by the application of the seventh and third rules. The final rule of Figure VI-5 is the rule for blocks, which indicates that the symbol sequences resulting from application of the rules to the statements within the block are simply to be concatenated in the order in which their corresponding statements appear in the block. This is natural, given the overall statement-by-statement nature of the rule application procedure.

The rules for the DC DYMOL SEND and RECEIVE statements, the first two rules of Figure VI-5, are clearly the most complex of the set. They also generate far more symbols per application than the other rules, since they involve unions over potentially quite sizable sets. In particular, the SEND rule specifies a union over the set of all distinct message types defined for the subject model, while the RECEIVE rule's unions are over that set and the set of all the model's outbound ports. Simplifications can frequently be realized in the application of these two rules, however, which significantly reduce the number of symbols generated. When it can be determined, by inspection or through techniques such as flow analysis [Alle76], that only certain messages could possibly be in a process' buffer (in the case of the first rule) or that only a certain subset of links or

messages could possibly figure in a particular RECEIVE operation (in the case of the second rule), then the unions specified in those rules need only be over the relevant possibilities. A common instance in which simplification can be performed occurs when a SEND instruction is immediately preceded by a buffer assignment statement. In such cases the union of the first rule collapses down to a single possibility and the rule produces only a single three symbol sequence like "#4' msg @9". Additional examples of simplifications in the application of these two rules appear in the example of the next section. Of course, the use of such simplifications is never required -- the algorithm always produces an appropriate DC constrained expression when no simplifications are employed. Indeed, the interpreted language represented is the same whether or not simplifications are used, since the simplifications merely eliminate symbols corresponding to impossible behaviors. However, the simplifications do often dramatically reduce the complexity of the derived DC constrained expression and can therefore be valuable additional tools in the derivation procedure.

Generation of the k subexpressions PEXPi as described in the preceding paragraphs completes the derivation of a DC constrained expression from the DPMS model M[DC] of a parallel system with dynamic connectivity. The interpreted language, L, represented by this DC constrained expression

with respect to the constraint set CS described in the previous section then corresponds to all the possible message behaviors of the modelled system, a fact which we shall argue further in a subsequent section. We conclude the present section with a few comments regarding DC constrained expressions and the derivation algorithm -- additional remarks on these topics are scattered throughout the following sections.

Several things may be noted about the DC constrained expressions generated by the algorithm which has just been described. For instance, the procedure does not directly apply any transformations to either the statement labels or the block brackets BEGIN and END of a DC DYMOL program. This reflects the fact that these parts of the DYMOL program have no direct behavioral significance -- labels being used only for the location counter values of process state representations and the block brackets serving only to combine statements into larger syntactic entities. Another observation is that the transformation rules for infinite iteration and nondeterministic indefinite iteration, i.e., the sixth and seventh rules of Figure VI-5, are identical. This situation follows from our observation of Chapter III that the word 'infinite' in our infinite iteration notion is used in a purely figurative sense, as no realistic computation actually continues forever. The correspondence between event expression operators and the basic control

constructs of programming, i.e., the correspondence of

$\sqcup$ with alternatives (conditionals)

* with repetition (iterations) and

concatenation with sequencing (blocks)

should also be noted. This correspondence is central to the relationship between constrained expressions and the possible message transmission behaviors of a DC DPMS model. Finally, the transformation rules clearly reflect the fact that SEND and RECEIVE are the two DC DYMOL instructions responsible for message transmission activity in DPMS-modelled systems, since these statements are the only ones which can lead to the appearance of symbols from E in the derived constrained expression. In the context of message transmission behavior, all the other statements serve only to constrain the sequencing and outcome of SEND and RECEIVE executions, as evidenced by the fact that their transformation rules generate only symbols from S.

An additional observation may be made regarding the relationship between DC constrained expressions and counter expressions. In fact, from a formal languages viewpoint the two types of expressions are equivalent, as shown by the following:

> Theorem VI.1: Any interpreted language L over an alphabet E which can be represented by a DC constrained expression can also be represented by a counter expression.

Proof: Momentarily disregarding the S2* and S3* terms of C1, a counter expression's interpreted language LC is defined as[1]:

$$LC = H(R' \sqcap C1)$$

where R' is a regular expression over $E \sqcup S1$. The definition of the interpreted language L of a DC constrained expression is:

$$L = H(R \sqcap C1 \sqcap C2 \sqcap C3)$$

where R is a regular expression over $E \sqcup S$. Now define a homomorphism $H': E \sqcup S \rightarrow E \sqcup S1$ by:

$$H'(x) = x \qquad \text{for } x \text{ in } E \sqcup S1$$

$$H'(x) = \lambda \qquad \text{for } x \text{ in } S2 \sqcup S3$$

Then, since C2 and C3 are shuffle expressions and therefore regular, and since regular sets are closed under intersection and homomorphism,

$$R'' = H'(R \sqcap C2 \sqcap C3)$$

is a regular expression over $E \sqcup S1$. Now clearly

$$H(R \sqcap C1 \sqcap C2 \sqcap C3) = H(H'(E \sqcap C2 \sqcap C3) \sqcap C1)$$

(note that the S2* and S3* terms of C1 drop out in evaluating the expression on the right) and thus

$$L = H(R'' \sqcap C1)$$

where R'' is a regular expression over $E \sqcup S1$.

---

[1]The additional @i* term of C1 not found in Welter's original constraining language simply permits the representation of computations in which not all sent messages are received. Thus our definition does not differ significantly from that given in [Welt76].

Therefore, any interpreted language represented by a constrained expression can also be represented by a counter expression, as asserted.

This result confirms the intuitive feeling that it should be possible to simulate the behavior of a DC system, with its fixed process structure and limited channel structure dynamicity, using a statically-structured model and some finite state encoding of the DC system's connectivity.

A DC Constrained Expression Example

Having described DC constrained expressions and an algorithm for deriving them from DPMS models, we now consider an example illustrating both of these topics. The example is also intended to indicate a possible role for these expressions and their derivation algorithm in the analysis of designs for certain classes of dynamically-structured, concurrent software systems.

In this example we will apply the DC constrained expression derivation algorithm to the producer-consumer system model of Figure VI-3. As the first step in this procedure, we define the mappings K, M, N, P, MP, PN and KM as shown in Figure VI-6. It should be clear that these definitions fulfill the requirements given for the various mappings in the derivation algorithm.

K	
K(producer1)	= 1
K(consumer1)	= 2
K(consumer2)	= 3
K(c_pool1)	= 4

M	
M(ready)	= 1
M(goods)	= 2
M(term)	= 3
M(done)	= 4
M(got_it)	= 5
M(Ø)	= 6

N	
N(producer1.ok)	= 1
N(consumer1.in)	= 2
N(consumer2.in)	= 3
N(c_pool1.pr)	= 4

P	
P(producer1.info)	= 1
P(producer1.cp)	= 2
P(consumer1.conf)	= 3
P(consumer2.conf)	= 4
P(c_pool1.cset)	= 5

MP(x,y)	= 5(x-1)+y	1≤x≤6	1≤y≤5
PN(x,y)	= 4(x-1)+y	1≤x≤5	1≤y≤4
KM(x,y)	= 6(x-1)+y	1≤x≤4	1≤y≤6

Mappings for the Example Derivation

Fig. VI-6

The second step in this derivation is the determination of the initial subexpression INIT. The initial complete link state L<0> of Figure VI-3 indicates that all links in the model are initially empty, hence no symbols from S1 will appear in the INIT subexpression. The model's initial configuration matrix C<0>, however, shows that two interprocess communication paths exist initially in the model. Therefore, the symbols:[1]

$$@PN(P(producer1.cp),N(c\_pool1.pr)) = @^{8} \quad \text{and}$$

$$@PN(P(c\_pool1.cset),N(producer1.ok)) = @^{17}$$

will be added to the INIT subexpression. Finally, inspection of the model's initial complete process state Q<0> reveals that all process buffers are initially empty. Hence the symbols:

$$\#KM(1,6) = \#^{6}, \qquad \#KM(2,6) = \#^{12},$$

$$\#KM(3,6) = \#^{18}, \quad \text{and} \quad \#KM(4,6) = \#^{24}$$

must all appear in the INIT subexpression. Thus, the complete subexpression is the concatenation of these symbols from S, i.e.,

$$INIT = @^{8}\ @^{17}\ \#^{6}\ \#^{12}\ \#^{18}\ \#^{24}$$

The example model of Figure VI-3 has four instantiated processes. Therefore, the derived DC constrained expression corresponding to that model has four subexpressions, PEXP1,

[1]To improve readability, we use a superscript notation for the numerical 'subscripts' of symbols in S throughout this example.

PEXP2, PEXP3 and PEXP4, whose interleave is concatenated to the INIT subexpression to form the complete DC constrained expression. These four subexpressions appear in Figure VI-7. We will describe the derivation procedure leading to one of these subexpressions, PEXP2, in detail, and simply point out interesting aspects of the other three derivations.

The PEXP2 subexpression corresponds to the process of the example model whose process identifier is mapped to 2 by the K mapping, i.e., process consumer1. Therefore, the derivation of PEXP2 is based upon the application of the rules of Figure VI-5 to the consumer process template of Figure VI-1 with appropriate substitutions for self-references. The procedure goes as follows:

The first statement (c1) of the consumer process template is an infinite iteration. The sixth rule of Figure VI-5 indicates that the transformation for an infinite iteration statement is simply to apply the '*' operator to the transformed version of its embedded statement. In this instance, the embedded statement is the entire remainder of the DYMOL program. Hence, application of the rule generates the outermost set of parentheses and the final '*' of the subexpression PEXP2.

The second statement of the consumer process template is 'RECEIVE in'. According to the second rule of Figure VI-

PEXP1:
(#1 #1' ready @2

(ø9 @13' ready ø9' #1 ⊔ ø13 @4' ready ø13' #1

⊔ ø17 @5' ready ø17' #1 ⊔ ø9 @23' got_it @9' #5

⊔ ø13 @24' got_it ø13' #5 ⊔ ø17 @25' got_it ø17' #5)

(#2 #2' goods @6

(ø9 @13' ready ø9' #1 ⊔ ø13 @4' ready ø13' #1

⊔ ø17 @5' ready ø17' #1 ⊔ ø9 @23' got_it @9' #5

⊔ ø13 @24' got_it ø13' #5 ⊔ ø17 @25' got_it ø17' #5))*

#3 #3' term @11 #4 #4' done @17)*

PEXP2:
((ø2 @6' goods ø2' #8 ⊔ ø2 @11' term ø2' #9)

((#8' #11 #11' got_it @23) ⊔ λ))*

PEXP3:
((ø3 @6' goods ø3' #14 ⊔ ø3 @11' term ø3' #23)

((#14' #17 #17' got_it @24) ⊔ λ))*

PEXP4:
(((ø8 @2' ready ø8' #19) ⊔ (ø8 @17' done ø8' #22))

((ϕ2 ϕ9 ((#19' ready @5) ⊔ (#22' done @20))

(((ø8 @2' ready ø8' #19) ⊔ (ø8 @17' done ø8' #22)) ϕ2' ϕ9'))

⊔ (ϕ3 ϕ13 ((#19' ready @5) ⊔ (#22' done @20))

(((ø8 @2' ready ø8' #19) ⊔ (ø8 @17' done ø8' #22))

ϕ3' ϕ13'))))*

The PEXPi Subexpressions

Fig. VI-7

5 this statement could generate a union of twenty-five five-symbol sequences, since the example model has five distinct message types and five outbound ports. However, examination of the model indicates that only one outbound port, producer1.info, could ever be connected to the 'in' port of consumer1 by an interprocess communication channel. This is easily confirmed by inspecting the model's initial instantaneous configuration (no entries in the consumer1 column of C<0>) and the ESTABLISH commands of the model's templates. Similarly, consideration of the producer process template of Figure VI-1 tells us that the only types of messages which can ever by sent through port producer1.info are 'info' and 'goods', since each of the relevant SEND instructions is immediately preceded by a buffer assignment. Hence, we can simplify the transformation so as to only generate a union of two five-symbol sequences. The first of these is:

$$\not\varnothing PN(1,2)\ @MP(2,1)\text{'}\ \text{goods}\ \not\varnothing PN(1,2)\text{'}\ \#KM(2,2)$$

$$= \not\varnothing^2\ @^6\text{'}\ \text{goods}\ \not\varnothing^2\text{'}\ \#^8$$

since P(producer1.info)=1, N(consumer1.in)=2, M(goods)=2 and K(consumer1)=2. It should be noted that the choice of arguments for both the N and K mappings here represents a case of (implicit) self-reference substitution. The second five-symbol sequence is derived in a similar fashion using the other possible value (i.e., 'term') for the y variable of the transformation rule. Thus, with the exception of its initial parenthesis, the entire first line of the PEXP2

subexpression results from the application of this transformation rule to the statement labelled c2 in the consumer process template.

The third statement of the process template is a nondeterministic conditional without an ELSE clause. The tenth transformation rule applies here, its application generating the first and next-to-last parentheses of the second line of PEXP2, the union with λ, and the symbol #^{8}', since KM(2,2)=8. Here again we find an instance of implicit self-reference, this time determining the first argument to the KM mapping.

The remaining symbols of PEXP2 result from application of the appropriate transformation rules to the statement list which is the embedded statement of the conditional. Thus the #^{11} generated by the buffer assignment statement (c4) and the sequence:

#^{11}' got_it @^{23}

are concatenated as specified by the final rule of Figure VI-5, i.e., the block rule. The simplification of the SEND rule into a single sequence of three symbols is an example of collapsing the rule's union when the SEND is immediately preceded by a buffer assignment statement. Implicit self-references in the buffer usages of both statements and the outbound port usage of the SEND statement influence the choice of arguments to the KM and MP mappings used in the

application of each of these final two rules.

The generation of subexpression PEXP3 is almost identical to that for PEXP2. The sole difference between the two concerns the different values used in the various mappings in cases of self-reference. It is therefore not surprising that PEXP3 differs from PEXP2 only in a few of the 'subscripts' of its constraint alphabet symbols.

The derivation of PEXP1, of course, requires application of the transformation rules to the producer process template of Figure VI-1. Although the procedure is quite straightforward and similar to that detailed above for PEXP2, a few observations regarding this particular derivation are in order. First, the possibility of embedding a '*' operator within the scope of another '*' operator, here due to the nested indefinite iterations of the producer process template, should be noted. We also observe that each SEND instruction of the template generates only a three symbol sequence in PEXP1, since each is immediately preceded by a buffer assignment statement. The six-way unions which make up the bulk of the subexpression arise from the template's two RECEIVE instructions (p4 and p8) and a partial simplification. A reasonably easy analysis of the example model shows that the producer can only receive messages of type 'got_it' or type 'ready'. Similarly, it is evident that producer1 can only receive

from three possible ports, i.e., its inbound ports will never be connected to either of its two outbound ports. Instead of a union of twenty-five symbol sequences, these two facts allow us to limit each RECEIVE rule application to generating a union of only six sequences. Further analysis would reveal that additional simplification is possible, since only one type of message can be received from each of the three possible message sources. However, the extra terms only lengthen the subexpression. They do not permit any more strings to be included in the interpreted language L represented by the derived DC constrained expression.

Simplifications have also been employed in transforming the RECEIVE instructions of the c_pool process template (cp2, cp7 and cp13) during the derivation of subexpression PEXP4. Here again it is possible to limit the possible sources to one (producer1.cp) and the possible message types received to two ('ready' and 'done'). The indefinite conditional statement (cp3) whose embedded statements make up most of the DYMOL program in Figure VI-2 generates the union (whose '␣' is the first symbol of the fourth line in PEXP4) which is the bulk of the subexpression.

Given the five subexpressions then, the complete derived DC constrained expression is:

INIT (PEXP1 ◻ PEXP2 ◻ PEXP3 ◻ PEXP4)

which appears fully written out in Figure VI-8. Also shown

```
(∂8 ∅17 #6 #12 #18 #24)
   □
(#1 #1' ready ∂2
(∅9 ∂13' ready ∅9' #1 ⊔ ∅13 ∂4' ready ∅13' #1
 ⊔ ∅17 ∂5' ready ∅17' #1 ⊔ ∅9 ∂23' got_it ∂9' #5
 ⊔ ∅13 ∂24' got_it ∅13' #5 ⊔ ∅17 ∂25' got_it ∅17' #5)
(#2 #2' goods ∂6
(∅9 ∂13' ready ∅9' #1 ⊔ ∅13 ∂4' ready ∅13' #1
 ⊔ ∅17 ∂5' ready ∅17' #1 ⊔ ∅9 ∂23' got_it ∂9' #5
 ⊔ ∅13 ∂24' got_it ∅13' #5 ⊔ ∅17 ∂25' got_it ∅17' #5) )*
#3 #3' term ∂11 #4 #4' done ∂17)*
   □
((∅2 ∂6' goods ∅2' #8 ⊔ ∅2 ∂11' term ∅2' #9)
 ((#8' #11 #11' got_it ∂23) ⊔ λ) )*
   □
((∅3 ∂6' goods ∅3' #14 ⊔ ∅3 ∂11' term ∅3' #23)
 ((#14' #17 #17' got_it ∂24) ⊔ λ) )*
   □
(((∅8 ∂2' ready ∅8' #19) ⊔ (∅8 ∂17' done ∅8' #22))
((∅2 ∅9 ((#19' ready ∂5) ⊔ (#22' done ∂20))
(((∅8 ∂2' ready ∅8' #19) ⊔ (∅8 ∂17' done ∅8' #22))∅2' ∅9'))
 ⊔ (∅3 ∅13 ((#19' ready ∂5) ⊔ (#22' done ∂20))
(((∅8 ∂2' ready ∅8' #19) ⊔ (∅8 ∂17' done ∅8' #22))
      ∅3' ∅13')) ) )*
```

```
C1 = ∂1* □ (∂1 ∂1')+ □ ... □ ∂25* □ (∂25 ∂25')
                □ S2* □ S3* □ E*
```

```
C2 = (∅1 (∅1* □ (∅1 ∅1'*)* ) ∅1')* □ ...
     □ (∅20 (∅20* □ (∅20 ∅20'*)* ) ∅20')*
                □ S1* □ S3* □ E*
```

```
C3 = (#1 #1'* ⊔ ... ⊔ #6 #6'*)
     □ (#7 #7'* ⊔ ... ⊔ #12 #12'*)
     □ (#13 #13'* ⊔ ... ⊔ #18 #18'*)
     □ (#19 #19'* ⊔ ... ⊔ #24 #24'*)
     □ S1* □ S2* □ E*
```

Complete Derived DC Constrained Expression and
Particularized Constraining Languages for the Example Model

Fig. VI-8

in Figure VI-8 are the DC constraining languages C1, C2 and C3 as particularized to this example.

The derived DC constrained expression of Figure VI-8 represents an uninterpreted language L', i.e., a set of strings over the augmented alphabet E ⊔ S. One subset of this uninterpreted language is described by the DC constrained expression of Figure VI-9. This latter constrained expression is the result of a particular set of choices regarding the ⊔'s and *'s of the Figure VI-8 expression and therefore contains no ⊔'s or *'s itself. The choices have produced a single string of symbols from each of the PEXPi subexpressions and thus the Figure VI-9 expression is simply the concatenation of the INIT subexpression (first line of the expression) with the interleave of the four symbol strings arising from these choices as applied to PEXP1 (second though seventh lines), PEXP2 (eighth line), PEXP3 (ninth and tenth lines) and PEXP4 (remaining lines). Consideration of the various subexpressions reveals that the particular set of choices resulting in the Figure VI-9 expression corresponds to a system behavior in which the producer sends each customer an information packet containing only one 'goods' message.

One of the strings of the subset of L' represented by Figure VI-9 is shown in Figure VI-10. This string, B', is the result of one possible interleaving chosen from among

```
(∂8 ∂17 #6 #12 #18 #24)
(#1 #1' ready ∂2 ∅17 ∂5' ready ∅17' #1
 #2 #2' goods ∂6 ∅9 ∂23' got_it ∅9' #5
 #3 #3' term ∂11 #4 #4' done ∂17
 #1 #1' ready ∂2 ∅17 ∂5' ready ∅17' #1
 #2 #2' goods ∂6 ∅13 ∂24' got_it ∅13' #5
 #3 #3' term ∂11 #4 #4' done ∂17)
□ (∅2 ∂6' goods ∅2' #8 #8' #11 #11' got_it ∂23 )
□ (∅3 ∂6' goods ∅3' #14 #14' #17 #17' got_it ∂24
   ∅3 ∂11' term ∅3' #15)
□ (∅8 ∂2' ready ∅8' #19 ∅2 ∅9 #19' ready ∂5
   ∅8 ∂17' done ∅8' #22 ∅2' ∅9'
   ∅8 ∂2' ready ∅8' #19 ∅3 ∅13 #19' ready ∂5
   ∅8 ∂17' done ∅8' #22 ∅3' ∅13')
```

A Subset of L' for the Derived Constrained Expression

Fig. VI-9

$B' =$

$\phi^{8}\ \phi^{17}\ \#^{6}\ \#^{12}\ \#^{18}\ \#^{24}$

$\#^{1}\ \#^{1'}$ ready $@^{2}$

$\phi^{8}\ @^{2'}$ ready $\phi^{8'}\ \#^{19}\ \phi^{2}\ \phi^{9}\ \#^{19'}$ ready $@^{5}$

$\phi^{17}\ @^{5'}$ ready $\phi^{17'}\ \#^{1}$

$\#^{2}\ \#^{2'}$ goods $@^{6}$

$\phi^{2}\ @^{6'}$ goods $\phi^{2'}\ \#^{8}\ \#^{8'}\ \#^{11}\ \#^{11'}$ got_it $@^{23}$

$\phi^{9}\ @^{23'}$ got_it $\phi^{9'}\ \#^{5}$

$\#^{3}\ \#^{3'}$ term $@^{11}\ \#^{4}\ \#^{4'}$ done $@^{17}$

$\phi^{8}\ @^{17'}$ done $\phi^{8'}\ \#^{22}\ \phi^{2'}\ \phi^{9'}$

$\#^{1}\ \#^{1'}$ ready $@^{2}$

$\phi^{8}\ @^{2'}$ ready $\phi^{8'}\ \#^{19}\ \phi^{3}\ \phi^{13}\ \#^{19'}$ ready $@^{5}$

$\phi^{17}\ @^{5'}$ ready $\phi^{17'}\ \#^{1}$

$\#^{2}\ \#^{2'}$ goods $@^{6}$

$\phi^{3}\ @^{6'}$ goods $\phi^{3'}\ \#^{14}\ \#^{14'}\ \#^{17}\ \#^{17'}$ got_it $@^{24}$

$\phi^{13}\ @^{24'}$ got_it $\phi^{13'}\ \#^{5}$

$\#^{3}\ \#^{3'}$ term $@^{11}\ \#^{4}\ \#^{4'}$ done $@^{17}$

$\phi^{8}\ @^{17'}$ done $\phi^{8'}\ \#^{22}$

$\phi^{3}\ @^{11'}$ term $\phi^{3'}\ \#^{15}\ \phi^{3'}\ \phi^{13'}$

$B = H(B' \sqcap C1 \sqcap C2 \sqcap C3)$

$=$ ready ready ready ready goods goods got_it got_it term done done ready ready ready ready goods goods got_it got_it term done done term

String B' Over $E \cup S$ and String B Over E

Fig. VI-10

the many described by the Figure VI-9 expression. In this particular instance,

$$B' \sqcap C1 \sqcap C2 \sqcap C3 \neq \emptyset$$

hence B' corresponds to some string B from the interpreted language L, i.e., a sequence of symbols from the alphabet E. The string B, which results from application of the H homomorphism to the intersection of B' and the languages of the constraint set CS, is shown at the bottom of Figure VI-10. For a DC constrained expression the alphabet E is just the set of distinct message types defined in the modelled system. Thus B is a string of messages from the producer-consumer model. In fact, as we show in the next section, it represents a possible message transmission behavior of the modelled system.

The message sequence represented by B corresponds closely to the expected sequence of message transmissions in the modelled system in a case where the producer sends two single-item information packets. A sequence of four[1] 'ready' messages would naturally appear as the producer process signalled c_pool of its intention to generate a packet and c_pool responded by indicating that a consumer

[1]Each completed communication of the modelled system generates two symbols in the message sequence describing its behavior. One represents the movement of the message from sender to link while the other signifies the message's movement from link to receiver. Under the semantics of DPMS, these events can be arbitrarily separated in time, hence the corresponding symbols need not, in general, occur adjacent to one another in the string.

was prepared to receive the information. The transmitted information ('goods') and the confirmation of its reception ('got_it') follow as expected, and finally the 'term' and 'done' messages indicate the completion of a packet transmission. The intermixing of the final four symbols of B (in fact, the derived constrained expression permits any ordering of the last three symbols in this behavioral representation) indicates that the receiving of the 'term' and 'done' messages by a consumer and the c_pool process, respectively, are potentially concurrent events.

Examination of the string B reveals one unexpected aspect of the modelled system's message transmission behavior, however. B contains only three 'term' symbols, indicating that one information packet termination indicator was not received by a consumer process, although it was sent by the producer. This is an unacceptable situation under our previously-stated assumption that a complete packet, including the termination indicator, should be received by a single consumer each time such a packet is generated. Yet the fact that B is a string in the interpreted language of the derived DC constrained expression indicates that this unacceptable behavior can be realized by the system as modelled in Figure VI-3. To the software system designer contemplating this DPMS model as a possible design for the producer-consumer system, this would presumably indicate that the proposed design was faulty.

In fact, examination of the constrained expression of Figure VI-8 can lead to discovery of the source of the difficulty. Since the ϕ^{2}' and ϕ^{3}' symbols of PEXP4 can be interleaved across the β^{2} and β^{3} symbols preceding the 'term' symbols in PEXP2 and PEXP3, respectively, we see that the manager can close a communication channel connecting consumer and producer before the consumer has retrieved the end marker for an information packet from the producer's info link. Reconsideration of the model's process templates (Figures VI-1 and VI-2) confirms that the producer's signal of completion is not necessarily correlated with a consumer's reception of the termination indicator, but only with the producer's sending of that indicator. Thus, the consumer pool manager is prematurely informed that a given producer-consumer transaction has been completed and may therefore erroneously prevent the consumer from obtaining the termination indicator by closing a communication channel too soon.

This example provides another illustration of the potential utility of the Dynamic Process Modelling Scheme in the design and analysis of dynamically-structured software systems. In this case, an algorithmic technique, DC constrained expression derivation, has been used to discover and isolate a mistake in a proposed software system design. Of course, the approach employed in analyzing the example model of chapter IV is also applicable to DC systems and

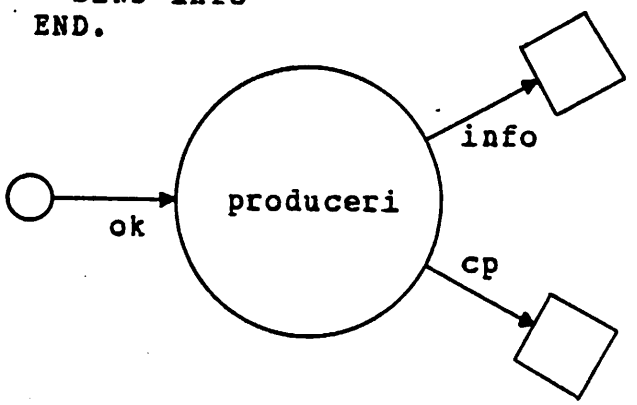
could have been used here as well.

We conclude this example by displaying a corrected version of the producer-consumer system model in Figures VI-11, VI-12 and VI-13. We give only the graphical representation of the revised model in Figure VI-13; the formal version is identical to that shown in Figure VI-3 except for an obvious modification to L<0> and the addition of (cp,cs) entries to C<0>(consumer1,c_pool1) and C<0>(consumer2,c_pool1). The modification made to the model here is essentially to have the consumer (at new statements c6 and c7), rather than the producer (at deleted statements p11 and p12), inform the consumer pool manager when a transaction has been completed. Readers are invited to exercise their understanding of the DC constrained expression derivation algorithm by applying it to the revised model in order to substantiate our assertion that the erroneous behavior has been eliminated.

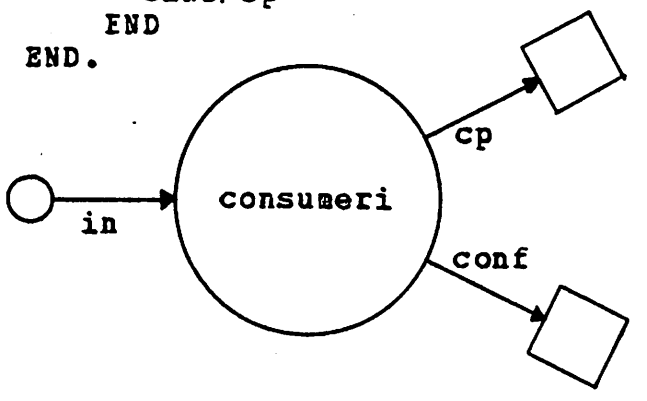
Correspondence Between DC Constrained Expressions and the Message Transmission Behavior of DPMS Models

In previous sections of this chapter we have stated that the strings of symbols in E* represented by a derived DC constrained expression correspond to the possible message transmission behaviors of the DPMS model from which the expression was derived. In this section we attempt to

```
producer: p1: WHILE INTERNAL TEST DO
                BEGIN
          p2:     SET BUFFER := ready;
          p3:     SEND cp;
          p4:     RECEIVE ok;
          p5:     WHILE INTERNAL TEST DO
                    BEGIN
          p6:         SET BUFFER := goods;
          p7:         SEND info;
          p8:         RECEIVE ok
                    END;
          p9:     SET BUFFER := term;
         p10:     SEND info
                END.
```



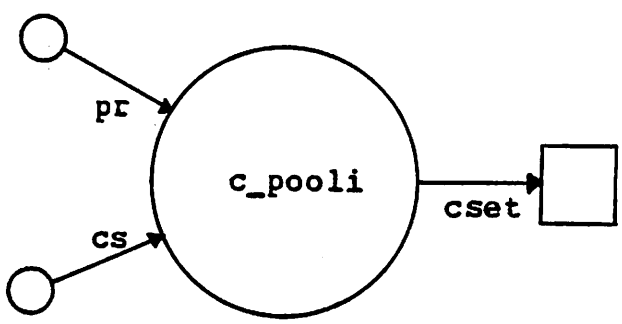

```
consumer: c1: DO FOREVER
                BEGIN
          c2:     RECEIVE in;
          c3:     IF BUFFER = goods THEN
                    BEGIN
          c4:         SET BUFFER := got_it;
          c5:         SEND conf
                    END
                  ELSE
                    BEGIN
          c6:         SET BUFFER := done;
          c7:         SEND, cp
                    END
                END.
```



Revised Producer and Consumer Process Templates

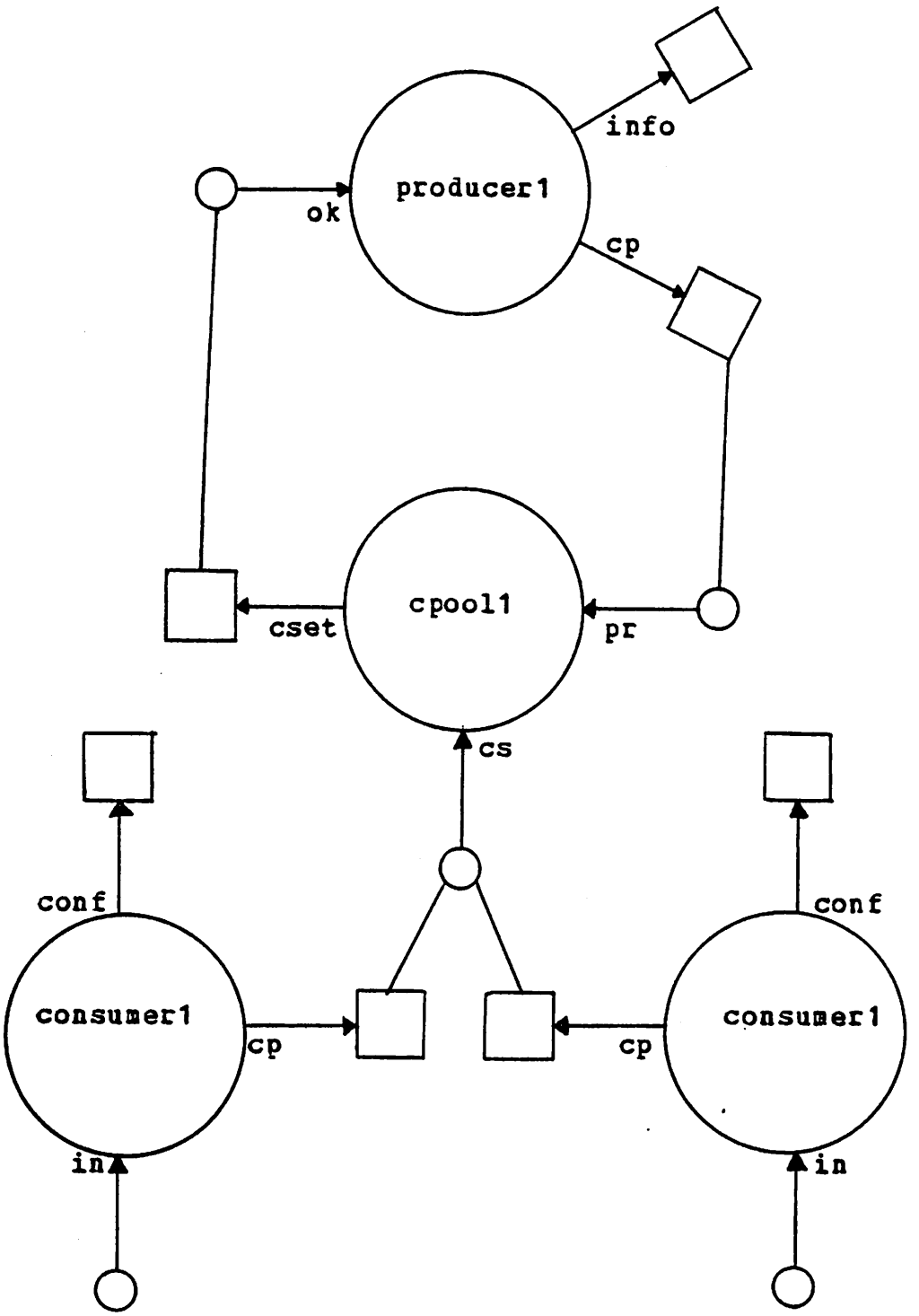
Fig. VI-11

```
c_pool: cp1: DO FOREVER
               BEGIN
        cp2:     RECEIVE pr;
        cp3:     IF INTERNAL TEST THEN
                   BEGIN
        cp4:         ESTABLISH producer1.info consumer1.in;
        cp5:         ESTABLISH consumer1.conf producer1.ok;
        cp6:         SEND cset;
        cp7:         RECEIVE cs;
        cp8:         CLOSE producer1.info consumer1.in;
        cp9:         CLOSE consumer1.conf producer1.ok
                   END
                  ELSE
                   BEGIN
       cp10:         ESTABLISH producer1.info consumer2.in;
       cp11:         ESTABLISH consumer2.conf producer1.ok;
       cp12:         SEND cset;
       cp13:         RECEIVE cs;
       cp14:         CLOSE producer1.info consumer2.in;
       cp15:         CLOSE consumer2.conf producer1.ok
                   END
                END.
```



Revised Consumer Pool Manager Process Template

Fig. VI-12



Revised Example DC System Model

Fig. VI-13

substantiate that claim. In fact, the remainder of this section is essentially a proof of the following:

> Theorem VI.2: Given a DPMS model M of a parallel system with dynamic connectivity, a DC constrained expression can be derived from that model such that a given message sequence is a possible message transmission behavior of M if and only if it is a string in the interpreted language L represented by the derived DC constrained expression.

Naturally, the import of this theorem depends heavily upon the meaning of "possible message behavior" in the preceding statement. The phrase is intended to connote an ordered sequence of message movements, both from a sender to a link and from a link to a recipient, which could accompany a possible computation of a DPMS model. We do, however, limit our interpretation to include only those computations in which each process has completely executed its DYMOL program some acceptable number (possibly including zero) of times, i.e., completed execution of the <statement> which is its process template. This interpretation permits any number of repetitions for any part of a DYMOL program, including the complete process template, which is an indefinite or infinite iteration. It does not, however, admit computations in which one or more processes have halted at some point in the middle of their DYMOL-specified

activity. Message transmission behaviors corresponding to particular instances of such partial computations may be considered, of course, by looking at prefixes of strings in L or by redefining process templates in M (e.g., truncating or expanding the DYMOL programs) such that their completion leaves processes at the appropriate stage of activity.

In showing that the derived DC constrained expression represents exactly the possible message transmission behaviors of M, we must show first that it represents at least all such behaviors, and then that it represents no more than those behaviors. That is, if MB is the set of all possible message behaviors for M and L is the interpreted language represented by the derived DC constrained expression, we wish to show first that MB $\leq$ L and then that L $\leq$ MB, since we can conclude from this that MB = L.

The message transmission behaviors of M necessarily result from the execution of SEND and RECEIVE instructions in the DYMOL-specified activity of M's processes. Thus the entire set of possible message transmission behaviors relative to a given process of M is the set of message sequences which would result from all its SEND and RECEIVE statements being executed in their specified order with all possible buffer and link contents. That is, considering each SEND execution as possibly moving any one of M's message types into some link and each RECEIVE as potentially

moving any type of message from a link to the process' buffer clearly covers all the possibilities for a given process' message transmission behaviors. The entire set of conceivable message transmission behaviors for M is then simply all possible interleavings of all possible combinations of these potential message transmission behaviors of the individual processes. Since the derived DC constrained expression for M is:

$$\text{INIT} \left(\left\| _{i} \right. \text{PEXPi} \right)$$

and INIT contributes no symbols from E, our problem reduces to showing that the individual subexpressions PEXPi represent the complete range of message transmission behavior possibilities for their respective processes.

Consideration of the first two transformation rules of Figure VI-5 shows that every SEND statement and RECEIVE statement in the DYMOL template corresponding to the jth process of M generates a union over all the message types of M in the subexpession PEXPj. Thus if the remaining transformation rules lead to a correct sequencing of these symbols, i.e., enough occurrences of these unions in the appropriate orders, and if the constraints imposed by symbols from S do not eliminate any sequences corresponding to possible message transmission behaviors of the jth process, we may conclude that PEXPj at least covers all the possibilities for that process' message transmission

behaviors.

That the derivation algorithm yields a correct sequencing of symbols follows from its statement-by-statement procedure, the block rule, and properties of the constraining language C3. The fact that no possible message transmissions are eliminated by the constraints also depends in part upon these same considerations. Since the derivation procedure and block rule produce symbol strings in the same order that the corresponding statements appear in a DYMOL process template, basic sequencing is certainly correctly represented in PEXPj. The previously-noted correspondence between event expression operators and programming language control constructs is reflected in the rules of Figure VI-5 and contributes further to appropriate sequencing. Finally, the data-dependencies of sequencing in a DC DYMOL program, i.e., the relationship between current buffer contents and the execution of embedded statements (which may lead to message transmissions) in conditional or indefinite iteration constructs, is captured in PEXPj by the use of symbols from S3 and the constraint imposed by the corresponding element, C3, of CS. The eighth, eleventh and twelfth rules of Figure VI-5 dictate that any symbol sequence corresponding to a DYMOL statement execution dependent upon the particular contents of the process' buffer will be prefixed by a symbol from S3 representing that dependency. Any instruction whose execution can alter

the buffer's contents will lead to the generation of a related S3 symbol (i.e., the second and fifth transformation rules) in PEXPj. The eventual intersection of PEXPj with C3 then eliminates any symbol sequences which do not appropriately reflect the dependency of statement execution sequencing on the contents of the process' buffer. The S3 symbols generated by the first transformation rule similarly lead to the elimination of symbol sequences for which the messages transmitted by SEND instructions do not depend upon the buffer contents at the time the SEND is executed. We therefore conclude that sequencing is appropriately represented in PEXPj and hence that MB $\leq$ L.

It remains to show that L does not include any message sequences which are not possible message transmission behaviors of M. Given the correctness of the individual PEXPi representations, this situation could only arise due to interprocess behavioral dependencies. In particular, the message transmissions corresponding to RECEIVE statement executions depend upon the existence of interprocess communication channels and the contents of links. If these dependencies are appropriately captured in the derived DC constrained expression, then we may conclude that MB = L.

We first consider the dependency of RECEIVEs on the existence of the appropriate communication channels. We have seen that a RECEIVE statement in the process template

for the jth process of M leads to a union over all possible message types in the subexpression PEXPj. The transformation rule producing that union also includes symbols from S2 surrounding each message symbol in the union. Now suppose that some string in L corresponded to a message sequence in which a RECEIVE has obtained a message 'msg' from a link to which it was not presently connected. This would imply that there was a string in the uninterpreted language L' in which some symbol sequence such as "ϕ^7 msg $\not\phi^{7'}$" appeared such that of the pair of symbols ϕ^7 and $\phi^{7'}$ its most recent predecessor in that string was $\phi^{7'}$, and that this string had survived intersection with CS to appear in L. But C2 only allows a sequence such as "ϕ^7 $\not\phi^{7'}$" to appear in a string in which it is preceded by the symbol ϕ^7 with no intervening occurrences of the symbol $\phi^{7'}$. Thus no such string may appear in L.

A similar argument applies to the dependency of RECEIVEs on the prior sending of an appropriate message to the appropriate link. The transformation placing a union over all message types into PEXPj for each RECEIVE statement in its process template also prefixes each such message symbol with a corresponding symbol from S1. Now suppose that some string in L corresponded to a message sequence in which a RECEIVE execution had been completed before a corresponding message had appeared in the appropriate link. This would imply that some string in L' contained a symbol

sequence such as "@^{s}' msg" which was not preceded by a matching occurrence of the symbol @^{s} and that this string had survived intersection with CS. But C1 clearly eliminates any such strings from the set which is mapped by H into the interpreted language L. Thus no such string can appear in L, and we have shown that L ≤ MB.

The preceding arguments have demonstrated that, for a given model's set of possible message transmission behaviors MB and the interpreted language L represented by its derived DC constrained expression, MB ≤ L and L ≤ MB. We therefore conclude that MB = L. In other words, the derived DC constrained expression represents exactly the set of possible message transmission behaviors of a DPMS-modelled parallel system with dynamic connectivity.

CHAPTER VII

SUMMARY AND CONCLUSIONS

In this dissertation we have investigated some of the issues involved in the modelling of parallel systems with dynamic structure. We have given a definition for this class of systems and indicated its significance as a characterization of certain complex, concurrent software system organizations. A formal modelling scheme for parallel systems with dynamic structure, the Dynamic Process Modelling Scheme, has been presented in full detail, and examples illustrating the modelling scheme and its potential as a tool for concurrent software system design have been discussed. The modelling scheme has been employed in investigating certain decidability questions for the class of parallel systems with dynamic structure. We have also introduced the constrained expressions technique for describing the behavioral properties of certain systems of this class. A subclass of parallel systems with dynamic structure, known as parallel systems with dynamic connectivity, has also been defined. The use of the formal modelling scheme in describing and analyzing this subclass has been discussed and an algorithm for deriving constrained

expressions representing the message transmission behavior of any system in this subclass was presented.

Conclusions

The class of parallel systems with dynamic structure poses some significant problems for the would-be modeller. Previous modelling schemes applicable to concurrent computational systems have either made no provision for dynamic structure or have handled it only for special cases or in particular contexts. Thus the definition of a general modelling scheme for this class of systems represents a significant contribution to the field of formal modelling of parallel systems.

The scheme was developed in response to a perceived need for this modelling capability. In particular, concurrent software systems are often organized as parallel systems with dynamic structure. It was hoped that the development of a formal modelling scheme for this class of systems would contribute to our understanding of their properties and eventually provide a basis for design and analysis methods applicable to dynamically-structured software systems.

Whether the Dynamic Process Modelling Scheme will prove useful in attacking the problems of designing and analyzing

dynamically-structured, concurrent software systems remains to be seen. The examples discussed in this dissertation suggest that it may. The ability to rigorously describe an intended design in an abstract manner and to consider its behavioral possibilities long before any implementation effort is undertaken would appear likely to facilitate the software design process. The Dynamic Process Modelling Scheme provides precisely these capabilities to the designer who models a proposed design using DPMS. However, the fact that many interesting questions regarding the behavior of DPMS models do not admit of general algorithmic solution, as we proved in Chapter V, could negatively impact the potential role of DPMS in software design and analysis. Further research is called for in determining just how serious a drawback these undecidability results represent. Our constrained expressions technique for parallel systems with dynamic connectivity suggests one possible avenue for circumventing undecidability -- the discovery of methods applicable to particular subclasses of parallel systems with dynamic structure. Other approaches, such as feedback analysis and simulation methods, also merit investigation.

The Dynamic Process Modelling Scheme and the class of systems which it can be used to describe are interesting in and of themselves. The systems represent a class of formal structures whose properties can be studied and whose relationship to other formal systems has received only

preliminary consideration here. The modelling scheme provides a tool for investigating questions about parallel systems with dynamic structure and a facility for formulating rigorous, abstract descriptions of real-world systems which fall into this class. The challenge remaining is to particularize its application to the problems which motivated its development -- the design and analysis of dynamically-structured, concurrent software systems.

Possibilities for Further Investigation

There are two basic directions along which we expect to pursue future research into the modelling of parallel systems with dynamic structure. One of these involves efforts to apply the results discussed in this dissertation to the problems of software design and analysis which initially led us to undertake this study. The actual usefulness of the Dynamic Process Modelling Scheme and of constrained expressions, and the practical significance of the undecidability results, ought now to be tested in a reasonably realistic setting. The other path for future exploration involves a continuation of work on the formal modelling of parallel systems with dynamic structure. Alternative modelling schemes as well as additional properties of the techniques presented here deserve examination.

In attempting to apply our work to the practical problems of software design and analysis we immediately encounter the question of how best to implement our modelling scheme and related techniques. One possibility may be to provide a stand-alone, simulation-based design aid similar to that described in [Sang77]. The well-defined formal structure underlying the Dynamic Process Modelling Scheme makes it very amenable to such an implementation, which could nearly be realized simply by automating the instantaneous configurations and DYMOL semantics defined in Chapter IV. Numerous questions of potential interest to a designer of dynamically-structured software systems, such as mean and maximum number of instantiated processes, mean, minimum and maximum process lifetime, distribution of instantiated processes by class and distribution of message transmission activity could be answered using such a system. An alternative possibility might be to use the concepts developed here as the basis for a design notation and set of analysis techniques expressly applicable to dynamically-structured software within a larger and more comprehensive design aid system. This approach has been tentatively explored and it is anticipated that dynamic process modelling concepts will be incorporated into future versions of the DREAM software design aid system [Ridd77a].

A related question concerns the appropriate strategies for use in automated versions of the analysis techniques

suggested by our work here. The automated generation of computations for a given DPMS model, for instance, might be based upon assumptions of pure randomness, or be goal-directed (e.g., attempting to reach a terminal instantaneous configuration) or perhaps user-directed in an interactive environment. The feasibility and efficacy of these various possibilities for assisting a software designer would be interesting to explore. Similarly, the most useful way of employing the often long and complex, but automatically derivable, constrained expressions described in Chapter VI is yet to be determined. The expression itself could be presented to a designer for perusal. Alternatively, strings from the interpreted language L could be generated from the expression, using one or more strategies of the sort suggested above for the automatic generation of computations, and presented to the designer as sample behaviors. Still another possibility is a system which would attempt to determine whether a designer-specified string could be an element of the interpreted language represented by a derived constrained expression. The feasibility of these latter two possibilities is a particularly interesting issue for future study.

A final question regarding practical application of this work concerns the naturalness of the modelling scheme as a software design description technique. The DPMS primitives such as messages, links, ports, buffers and

templates could conceivably prove unwieldy for a software system designer. A different set of basic constructs might be defined, which would be more natural to the practitioner, yet could be mapped onto DPMS for simulation and/or analysis. These considerations seem to demand some empirical data and suggest experimental implementations of the Dynamic Process Modelling Scheme.

Continued investigations into the formal modelling of parallel systems with dynamic structure could focus on any number of unresolved questions remaining within the Dynamic Process Modelling Scheme. The extended version of DYMOL defined in Chapter III, for instance, should be studied to see if its descriptive power is significantly greater than that of basic DYMOL. It would also be interesting to know whether there exists any reasonably straightforward translation of the extended version into the version which we have considered in this dissertation. The notion of constrained expressions also seems worthy of further attention. Their applicability to behavioral descriptions other than message transmission should be examined and their extension to other PSDS subclasses considered. Preliminary work on this latter question indicates that a more complicated class of constrained expressions can be applied to a PSDS subclass called parallel systems with bounded dynamic structure (BDS) when certain appropriate restrictions are made on the usage of links and DYMOL set

variables. Further work on the BDS case in particular and constrained expressions in general is definitely anticipated for the near future. Finally, completely different approaches to analyzing DPMS models remain to be discovered. One particularly intriguing possibility in this regard is a treatment based upon the properties of a model's configuration matrices, such as their size, shape, sparseness and pattern of non-null entries. Certain general properties of these matrices could indicate specific behavioral characteristics of the modelled system. Many other potential analysis techniques for DPMS models might also warrant consideration.

We would not expect, however, that inquiry into the formal issues of modelling parallel systems with dynamic structure will be limited to work on the Dynamic Process Modelling Scheme. Alternative formulations could well prove more suitable to practical application or more conducive to formal scrutiny and definitely should be sought. One such alternative, a dynamically-structured version of Petri nets, has received some preliminary attention in the course of our research and represents a potentially interesting approach to PSDS modelling. Although this initial investigation did not produce a formalism which adequately captured the PSDS features of greatest interest for software system design and analysis, and hence was abandoned in favor of the DPMS inquiry, further effort in this area could well prove

fruitful.

As this brief survey of possibilities suggests, the range of potential research topics in this area is indeed extensive. It, therefore, seems probable that this dissertation marks only an initial step in the investigation of the modelling of parallel systems with dynamic structure.

APPENDICES

APPENDIX A

BNF DESCRIPTION OF

BASIC DYMOL SYNTAX

```
<process template> ::= <process class id> : <statement> .

<statement> ::= <simple statement> |
                <label> : <iterative statement> |
                <label> : <selection statement> |
                <label> : <conditional statement>

<simple statement> ::= <block> |
                       <label> : <basic statement>

<block> ::= BEGIN <statement list> END

<statement list> ::= <statement> |
                     <statement list> ; <statement>

<basic statement> ::= <create statement> |
                      <destroy statement> |
                      <establish statement> |
                      <close statement> |
                      <send statement> |
                      <receive statement> |
                      <assignment statement>

<iterative statement> ::= <infinite iterative statement> |
                          <indefinite iterative statement>

<infinite iterative statement> ::= DO FOREVER <statement>

<indefinite iterative statement> ::=
                       WHILE <condition> DO <statement>

<selection statement> ::= <partial selection statement> |
                          <total selection statement>

<partial selection statement> ::=
                FOR SOME <value selection> DO <statement>

<total selection statement> ::=
                FOR ALL <value selection> DO <statement>

<value selection> ::= <assignment>
```

```
<conditional statement> ::=
  IF <condition> THEN <simple statement> ELSE <statement> |
  IF <condition> THEN <statement>

<condition> ::= INTERNAL TEST |
                <buffer condition> |
                <configuration condition>

<buffer condition> ::= BUFFER = <message class>

<configuration condition> ::= <process id expression> IN A

<create statement> ::=
            CREATE <process class id> <process ref var>

<destroy statement> ::= DESTROY <process id expression>

<establish statement> ::= ESTABLISH <port name> <port name>

<close statement> ::= CLOSE <port name> <port name>

<send statement> ::= SEND <port id>

<receive statement> ::= RECEIVE <port id>

<assignment statement> ::= <buffer assignment statement> |
                           <assignment>

<buffer assignment statement> ::=
                           SET BUFFER := <message class>

<assignment> ::= <var id> :- <var id> |
                 <var id> := <expression>

<process id> ::= <process class id> <integer>

<process id expression> ::= <process id> |
                            <process ref var> |
                            ME

<port name> ::= <process id expression> . <port id>

<process class id> ::= <char> |
                       <simple id> <char>

<message class> ::= <simple id>

<process ref var> ::= <simple id>

<label> ::= <simple id>

<port id> ::= <simple id>
```

```
<var id> ::= <simple id> |
             <set var id>

<simple id> ::= <char> |
                <simple id> <char or digit>

<set var id> ::= / <simple id> /

<expression> ::= <term> |
                 <term> + <expression> |
                 <term> - <expression> |
                 ( <expression> )

<term> ::= <var id> |
           <set specification> |
           A

<set specification> ::= { <item list> }

<item list> ::= <item> |
                <item list> , <item>

<item> ::= <process class id> |
           <process id> |
           <port name> |
           <port id>

<char> ::= a | b | c | d | ... | x | y | z | # | _

<integer> ::= <digit> |
              <integer> <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<char or digit> ::= <char> |
                    <digit>
```

APPENDIX B

BNF DESCRIPTION OF

EXTENDED DYMOL SYNTAX

```
<process template> ::=
            <process template header> : <statement> .

<process template header> ::= <process class id> |
                  <process class id> ( <parameter list> )

<parameter list> ::= <local param id> |
                     <parameter list> , <local param id>

<statement> ::= <simple statement> |
                <label> : <iterative statement> |
                <label> : <selection statement> |
                <label> : <conditional statement>

<simple statement> ::= <block> |
                       <label> : <basic statement>

<block> ::= BEGIN <statement list> END

<statement list> ::= <statement> |
                     <statement list> ; <statement>

<basic statement> ::= <create statement> |
                      <destroy statement> |
                      <establish statement> |
                      <close statement> |
                      <send statement> |
                      <receive statement> |
                      <assignment statement>

<iterative statement> ::= <infinite iterative statement> |
                          <indefinite iterative statement>

<infinite iterative statement> ::= DO FOREVER <statement>

<indefinite iterative statement> ::=
                        WHILE <condition> DO <statement>

<selection statement> ::= <partial selection statement> |
                          <total selection statement>

<partial selection statement> ::=
                FOR SOME <value selection> DO <statement>
```

```
<total selection statement> ::=
                FOR ALL <value selection> DO <statement>

<value selection> ::= <assignment>

<conditional statement> ::=
  IF <condition> THEN <simple statement> ELSE <statement> |
  IF <condition> THEN <statement>

<condition> ::= INTERNAL TEST |
                <buffer condition> |
                <configuration condition>

<buffer condition> ::= BUFFER = <message class>

<configuration condition> ::= <process id expression> IN A

<create statement> ::=
    CREATE <process class specification> <process ref var>

<destroy statement> ::= DESTROY <process id expression>

<establish statement> ::=
  ESTABLISH <port name expression> <port name expression>

<close statement> ::=
      CLOSE <port name expression> <port name expression>

<send statement> ::= SEND <port id>

<receive statement> ::= RECEIVE <port id>

<assignment statement> ::= <buffer assignment statement> |
                           <assignment>

<buffer assignment statement> ::=
                SET BUFFER := <message class expression>

<assignment> ::= <var id> := <var id> |
                 <var id> := <expression>

<process id> ::= <process class id> <integer>

<process id expression> ::= <process id> |
                            <process ref var> |
                            ME |
                            <local param id>

<process class specification> ::=
    <process class id expression> |
    <process class id expression> (<actual param list> )
```

```
<process class id expression> ::= <process class id> |
                                  <local param id>

<actual param list> ::= <actual param> |
                        <actual param list> , <actual param>

<actual param> ::= <process id expression> |
                   <process class expression> |
                   <message class expression> |
                   <port name expression>

<port name expression> ::= <port name> |
                           <local param id>

<port name> ::= <process id expression> . <port id>

<process class id> ::= <char> |
                       <simple id> <char>

<message class expression> ::= <message class> |
                               <local param id>

<message class> ::= <simple id>

<process ref var> ::= <simple id>

<label> ::= <simple id>

<port id> ::= <simple id>

<local param id> ::= <var id>

<var id> ::= <simple id> |
             <set var id>

<simple id> ::= <char> |
                <simple id> <char or digit>

<set var id> ::= / <simple id> /

<expression> ::= <term> |
                 <term> + <expression> |
                 <term> - <expression> |
                 ( <expression> )

<term> ::= <var id> |
           <local param id> |
           <set specification> |
           A

<set specification> ::= { <item list> }
```

```
<item list> ::= <item> |
                <item list> , <item>

<item> ::= <process class id> |
           <process id> |
           <port name> |
           <port id>

<char> ::= a | b | c | d | ... | x | y | z | # | _

<integer> ::= <digit> |
              <integer> <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<char or digit> ::= <char> |
                    <digit>
```

APPENDIX C

A COMPUTATION OF THE EXAMPLE MODEL OF CHAPTER IV

One possible computation of M is x<0>x<1>...x<30> where:

x<0> = (C<0>,A<0>,Q<0>,L<0>)

x<1> = (C<0>,A<0>,Q<1>,L<0>) with
Q<1> = ((sc2,tvar=Ø,/subs/=Ø,Ø),(sy1,Ø))

x<2> = (C<0>,A<0>,Q<2>,L<0>) with
Q<2> = ((sc3,tvar=Ø,/subs/=Ø,Ø),(sy1,Ø))

x<3> = (C<0>,A<0>,Q<3>,L<0>) with
Q<3> = ((sc3,tvar=Ø,/subs/=Ø,Ø),(sy2,Ø))

x<4> = (C<4>,A<4>,Q<4>,L<4>) with
A<4> = {sched1, synch1, subtask2}
Q<4> = ((sc4,tvar=<subtask2>,/subs/=Ø,Ø),
(sy2,Ø),(st1,Ø))
L<4> = ((<sem>),Ø)

x<5> = (C<5>,A<4>,Q<5>,L<4>) with
Q<5> = ((sc5,tvar=<subtask2>,/subs/=Ø,Ø),
(sy2,Ø),(st1,Ø))

x<6> = (C<5>,A<4>,Q<6>,L<4>) with
Q<6> = ((sc5,tvar=<subtask2>,/subs/=Ø,Ø),
(sy2,Ø),(st2,Ø))

x<7> = (C<7>,A<4>,Q<7>,L<4>) with
Q<7> = ((sc6,tvar=<subtask2>,/subs/=Ø,Ø),
(sy2,Ø),(st2,Ø))

x<8> = (C<7>,A<4>,Q<8>,L<8>) with
Q<8> = ((sc6,tvar=<subtask2>,/subs/=Ø,Ø),
(sy2,Ø),(st3,sem))
L<8> = (Ø,Ø)

x<9> = (C<7>,A<4>,Q<9>,L<8>) with
Q<9> = ((sc2,tvar=<subtask2>,/subs/=<subtask2>,Ø),
(sy2,Ø),(st3,sem))

x<10> = (C<7>,A<4>,Q<10>,L<8>) with
Q<10> = ((sc3,tvar=<subtask2>,/subs/=<subtask2>,Ø),
(sy2,Ø),(st3,sem))

M = (P, x<0>, T)

where:

P = {sched, synch, subtask}

x<0> = (C<0>, A<0>, Q<0>, L<0>)

T = {(C,A,Q,L) | ∀i q[subtaski]=(st2,-), q[synch1]=(sy2,-), L=(∅,...,∅)}

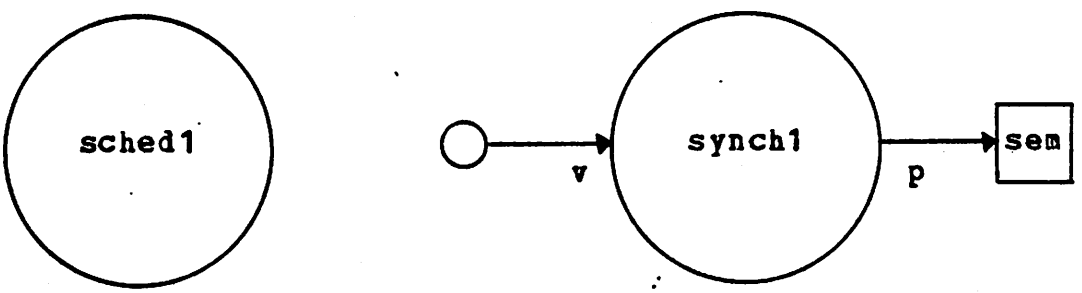
with:

C<0> =

	sched1	synch1
sched1	∅	∅
synch1	∅	∅

A<0> = {sched1, synch1} L<0> = ((<sem>))

Q<0> = ((sc1, tvar=∅, /subs/=∅, ∅), (sy1, ∅))



The Example Model M

Fig. C-1

C<4> =

	sched1	synch1	subtask2
sched1	∅	∅	∅
synch1	∅	∅	∅
subtask2	∅	∅	∅

C<5> =

	sched1	synch1	subtask2
sched1	∅	∅	∅
synch1	∅	∅	∅
subtask2	∅	(out,v)	∅

C<7> =

	sched1	synch1	subtask2
sched1	∅	∅	∅
synch1	∅	∅	(p,in)
subtask2	∅	(out,v)	∅

Configuration Matrices for the Sample Computation

Fig. C-2

```
x<11> = (C<11>,A<11>,Q<11>,L<11>) with
          A<11> = {sched1, synch1, subtask2, subtask7}
          Q<11> = ((sc4,tvar=<subtask7>,/subs/=<subtask2>,Ø),
                    (sy2,Ø),(st3,sem),(st1,Ø))
          L<11> = (Ø,Ø,Ø)

x<12> = (C<11>,A<11>,Q<12>,L<12>) with
          Q<12> = ((sc4,tvar=<subtask7>,/subs/=<subtask2>,Ø),
                    (sy2,Ø),(st1,sem),(st1,Ø))
          L<12> = (Ø,(<sem>),Ø)

x<13> = (C<13>,A<11>,Q<13>,L<12>) with
          Q<13> = ((sc5,tvar=<subtask7>,/subs/=<subtask2>,Ø),
                    (sy2,Ø),(st1,sem),(st1,Ø))

x<14> = (C<13>,A<11>,Q<14>,L<14>) with
          Q<14> = ((sc5,tvar=<subtask7>,/subs/=<subtask2>,Ø),
                    (sy3,sem),(st1,sem),(st1,Ø))
          L<14> = (Ø,Ø,Ø)

x<15> = (C<15>,A<11>,Q<15>,L<14>) with
          Q<15> = ((sc6,tvar=<subtask7>,/subs/=<subtask2>,Ø),
                    (sy3,sem),(st1,sem),(st1,Ø))

x<16> = (C<15>,A<11>,Q<16>,L<16>) with
          Q<16> = ((sc6,tvar=<subtask7>,/subs/=<subtask2>,Ø),
                    (sy1,sem),(st1,sem),(st1,Ø))
          L<16> = ((<sem>),Ø,Ø)

x<17> = (C<15>,A<11>,Q<17>,L<16>) with
          Q<17> = ((sc2,/subs/=<subtask2,subtask7>,
                    tvar=<subtask7>,Ø),(sy1,sem),(st1,sem),(st1,Ø))

x<18> = (C<15>,A<11>,Q<18>,L<16>) with
          Q<18> = ((sc2,/subs/=<subtask2,subtask7>,
                    tvar=<subtask7>,Ø),(sy2,sem),(st1,sem),(st1,Ø))

x<19> = (C<15>,A<11>,Q<19>,L<16>) with
          Q<19> = ((sc3,/subs/=<subtask2,subtask7>,
                    tvar=<subtask7>,Ø),(sy2,sem),(st1,sem),(st1,Ø))

x<20> = (C<15>,A<11>,Q<20>,L<16>) with
          Q<20> = ((sc3,/subs/=<subtask2,subtask7>,
                    tvar=<subtask7>,Ø),(sy2,sem),(st2,sem),(st1,Ø))

x<21> = (C<21>,A<21>,Q<21>,L<21>) with
          A<21> = {sched1,synch1,subtask2,subtask7,subtask9}
          Q<21> = ((sc4,/subs/=<subtask2,subtask7>,
                    tvar=<subtask9>,Ø),(sy2,sem),(st2,sem),
                    (st1,Ø),(st1,Ø))
          L<21> = ((<sem>),Ø,Ø,Ø)
```

C<11> =

	sched1	synch1	subtask2	subtask7
sched1	∅	∅	∅	∅
synch1	∅	∅	(p,in)	∅
subtask2	∅	(out,v)	∅	∅
subtask7	∅	∅	∅	∅

C<13> =

	sched1	synch1	subtask2	subtask7
sched1	∅	∅	∅	∅
synch1	∅	∅	(p,in)	∅
subtask2	∅	(out,v)	∅	∅
subtask7	∅	(out,v)	∅	∅

More Configuration Matrices

Fig. C-3

C<15> =

	sched1	synch1	subtask2	subtask7
sched1	∅	∅	∅	∅
synch1	∅	∅	(p,in)	(p,in)
subtask2	∅	(out,v)	∅	∅
subtask7	∅	(out,v)	∅	∅

C<21> =

	sched1	synch1	subtask2	subtask7	subtask9
sched1	∅	∅	∅	∅	∅
synch1	∅	∅	(p,in)	(p,in)	∅
subtask2	∅	(out,v)	∅	∅	∅
subtask7	∅	(out,v)	∅	∅	∅
subtask9	∅	∅	∅	∅	∅

Still More Configuration Matrices

Fig. C-4

```
x<22> = (C<21>,A<21>,Q<22>,L<21>) with
          Q<22> = ((sc4,/subs/=<subtask2,subtask7>,
                    tvar=<subtask9>,Ø),(sy2,sem),(st2,sem),
                    (st2,Ø),(st1,Ø))

x<23> = (C<23>,A<21>,Q<23>,L<21>) with
          Q<23> = ((sc5,/subs/=<subtask2,subtask7>,
                    tvar=<subtask9>,Ø),(sy2,sem),(st2,sem),
                    (st2,Ø),(st1,Ø))

x<24> = (C<23>,A<21>,Q<24>,L<21>) with
          Q<24> = ((sc5,/subs/=<subtask2,subtask7>,
                    tvar=<subtask9>,Ø),(sy2,sem),(st2,sem),
                    (st2,Ø),(st2,Ø))

x<25> = (C<25>,A<21>,Q<25>,L<21>) with
          Q<25> = ((sc6,/subs/=<subtask2,subtask7>,
                    tvar=<subtask9>,Ø),(sy2,sem),(st2,sem),
                    (st2,Ø),(st2,Ø))

x<26> = (C<25>,A<21>,Q<26>,L<26>) with
          Q<26> = ((sc6,/subs/=<subtask2,subtask7>,
                    tvar=<subtask9>,Ø),(sy2,sem),(st2,sem),
                    (st3,sem),(st2,Ø))
          L<26> = (Ø,Ø,Ø,Ø)

x<27> = (C<25>,A<21>,Q<27>,L<26>) with
          Q<27> = ((sc2,/subs/=<subtask2,subtask7,subtask9>,
                    tvar=<subtask9>,Ø),(sy2,sem),(st2,sem),
                    (st3,sem),(st2,Ø))

x<28> = (C<25>,A<21>,Q<28>,L<26>) with
          Q<28> = ((sc7,/subs/=<subtask2,subtask7,subtask9>,
                    tvar=<subtask9>,Ø),(sy2,sem),(st2,sem),
                    (st3,sem),(st2,Ø))

x<29> = (C<25>,A<21>,Q<29>,L<25>) with
          Q<29> = ((sc8,/subs/=<subtask2,subtask7,subtask9>,
                    tvar=<subtask9>,Ø),(sy2,sem),(st2,sem),
                    (st3,sem),(st2,Ø))

x<30> = (C<25>,A<21>,Q<30>,L<25>) with
          Q<30> = ((sc9,/subs/=<subtask2,subtask9>,
                    tvar=<subtask7>,Ø),(sy2,sem),(st2,sem),
                    (st3,sem),(st2,Ø))
```

C<23> =

	sched1	synch1	subtask2	subtask7	subtask9
sched1	∅	∅	∅	∅	∅
synch1	∅	∅	(p,in)	(p,in)	∅
subtask2	∅	(out,v)	∅	∅	∅
subtask7	∅	(out,v)	∅	∅	∅
subtask9	∅	(out,v)	∅	∅	∅

C<25> =

	sched1	synch1	subtask2	subtask7	subtask9
sched1	∅	∅	∅	∅	∅
synch1	∅	∅	(p,in)	(p,in)	(p,in)
subtask2	∅	(out,v)	∅	∅	∅
subtask7	∅	(out,v)	∅	∅	∅
subtask9	∅	(out,v)	∅	∅	∅

And Still More Configuration Matrices

Fig. C-5

APPENDIX D

BNF DESCRIPTION OF

DC DYMOL SYNTAX

```
<process template> ::= <process class id> : <statement> .

<statement> ::= <simple statement> |
                <label> : <iterative statement> |
                <label> : <conditional statement>

<simple statement> ::= <block> |
                       <label> : <basic statement>

<block> ::= BEGIN <statement list> END

<statement list> ::= <statement> |
                     <statement list> ; <statement>

<basic statement> ::= <establish statement> |
                      <close statement> |
                      <send statement> |
                      <receive statement> |
                      <assignment statement>

<iterative statement> ::= <infinite iterative statement> |
                          <indefinite iterative statement>

<infinite iterative statement> ::= DO FOREVER <statement>

<indefinite iterative statement> ::=
                       WHILE <condition> DO <statement>

<conditional statement> ::=
  IF <condition> THEN <simple statement> ELSE <statement> |
  IF <condition> THEN <statement>

<condition> ::= INTERNAL TEST |
                <buffer condition>

<buffer condition> ::= BUFFER = <message class>

<establish statement> ::= ESTABLISH <port name> <port name>

<close statement> ::= CLOSE <port name> <port name>

<send statement> ::= SEND <port id>
```

<receive statement> ::= RECEIVE <port id>

<assignment statement> ::= <buffer assignment statement>

<buffer assignment statement> ::=
SET BUFFER := <message class>

<process id> ::= <process class id> <integer>

<process id expression> ::= <process id> |
ME

<port name> ::= <process id expression> . <port id>

<process class id> ::= <char> |
<simple id> <char>

<message class> ::= <simple id>

<label> ::= <simple id>

<port id> ::= <simple id>

<simple id> ::= <char> |
<simple id> <char or digit>

<char> ::= a | b | c | d | ... | x | y | z | # | _

<integer> ::= <digit> |
<integer> <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<char or digit> ::= <char> |
<digit>

REFERENCES

REFERENCES

[Adam68] Adams, D. A. A computation model with data-flow sequencing. Ph.D. Th. and Tech. Rep. CS-117, Computer Sci. Dep., Stanford U., Stanford, Calif., Dec. 1968.

[Alle76] Allen, F. E., and Cocke, J. A program data flow analysis procedure. Comm. ACM, 19, 3 (Mar. 1976), 137-147.

[Baer70] Baer, J. L., Bovet, D. P., and Estrin, G. Legality and other properties of graph models of computation. J. ACM, 17, 3 (July 1970), 543-554.

[Bohm66] Bohm, C., and Jacopini, G. Flow diagrams, Turing machines and languages with only two formation rules. Comm. ACM 9, 5 (May 1966), 366-371.

[Bred70] Bredt, T. H. A survey of models for parallel computing. Digital Systems Lab. Tech. Rep. 8, Stanford U., Stanford, Calif., Aug. 1970.

[Brin70] Brinch Hansen, P. The nucleus of a multiprogramming system. Comm. ACM 13, 4 (Apr. 1970), 238-251, 250.

[Camp76] Campbell, R. H. Path expressions: A technique for specifying process synchronization. Ph.D. Th., U. of Newcastle upon Tyne, Newcastle upon Tyne, England, Aug. 1976.

[Cerf72] Cerf, V. G. Multiprocessors, semaphores, and a graph model of computation. Ph.D. Th. and Tech. Rep. ENG-7223, Computer Sci. Dep., U. of California, Los Angeles, April 1972.

[Denn66] Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. Comm. ACM 9, 3 (March 1966), 143-155.

[Dijk68] Dijkstra, E. W. Cooperating sequential processes. In Programming Languages, pp.43-112. Edited by F. Genuys. Academic Press, New York, 1968.

[Dijk72] Dijkstra, E. W. Notes on structured programming. In Structured Programming. Academic Press, New York, pp. 1-82.

[Fenn77] Fennel, R. D., and Lesser, V. R. Parallelism in AI problem solving: A case study of HEARSAY-II. IEEE Trans. Comp., C-26, 2, (Feb. 1977), 98-111.

[Feld77] Feldman, J. A. A programming methodology for distributed computing (among other things). Dept. of Comp. Sci., U. of Rochester, Rochester, N. Y., 1977.

[Floy67] Floyd, R. W. Assigning meaning to programs. Proc. Symp. in Appl. Math., Mathematical Aspects of Computer Science, vol. 19, Amer. Math. Soc. (1967), pp. 19-32.

[Fris71] von Frisch, K. Bees: Their Vision, Chemical Senses and Language. Rev. ed. Cornell University Press, Ithaca, 1971.

[Gins66] Ginsburg, S. The Mathematical Theory of Context-Free Languages. McGraw-Hill, New York, 1966.

[Hack75] Hack, M. Decision problems for Petri nets and vector addition systems. MIT Project MAC TM-59, March 1975.

[Hack76a] Hack, M. Decidability questions for Petri nets. Ph.D. Th. and Lab. for Computer Sci. Tech. Rep. 161, MIT, June 1976.

[Hack76b] Hack, M. Petri net languages. MIT Lab. for Computer Sci. TR-159, March 1976.

[Hans76] Hanson, A., Riseman, E., and Williams, T. Constructing semantic models in the visual analysis of scenes. Proc. IEEE Milwaukee Symp. on Automatic Computation and Control. April 1976, 97-102.

[Hara69] Harary, F. Graph Theory. Addison-Wesley, Reading, Mass., 1969.

[Harr76] Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. Protection in operating systems. Comm. ACM 19, 8 (Aug. 1976), 461-471.

[Hoar69] Hoare, C. A. R., An axiomatic basis for computer programming. Comm. ACM 12, 10 (Oct. 1969), 576-583.

[Horn73] Horning, J. J., and Randall, B. Process structuring. Computing Surveys 5, 1 (March 1973), 5-30.

[Hyma66] Hyman, H. Comment on a problem in concurrent programming control. Comm. ACM 9, 1 (Jan. 1966), 45.

[Jone73] Jones, A. K. Protection in programmed systems. Ph.D. Th., Dep. of Computer Sci., Carnegie-Mellon U., Pittsburgh, Pa., June 1973.

[Karp66] Karp, R. M., and Miller, R. E. Properties of a model for parallel computations: determinacy, termination, queuing. SIAM J. Appl. Math. 14, 6 (Nov. 1966) 1390-1411.

[Karp67] Karp, R. M., and Miller, R. E. Parallel program schemata: A mathematical model for parallel computation. IEEE Conf. Record of 1967 Eighth Ann. Symp. on Switching and Automata Theory, 1967.

[Karp69] Karp, R. M., and Miller, R. E. Parallel Program Schemata. J. Computer System Sci. 3, 4 (May 1969), 147-195.

[Kell76] Keller, R. M. Formal verification of parallel programs. Comm. ACM 19, 7 (July, 1976), 371-384.

[Knut69] Knuth, D. E. The Art of Computer Programming, Vol. 2. Addison-Wesley, Reading, Mass., 1969.

[Laue72] Lauer, H. C. Correctness in operating systems. Ph.D. Th., Dep. of Computer Sci., Carnegie-Mellon U., Pittsburgh, Pa., Sept. 1972.

[Lipt77] Lipton, R. J., and Snyder, L. A linear time algorithm for deciding subject security. J. ACM 24, 3 (July 1977), 455-464.

[Mann74] Manna, Z. Mathematical Theory of Computation. McGraw-Hill, New York, 1974.

[Mill73] Miller, R. E. A comparison of some theoretical models of parallel computation. IEEE Trans. Comp. C-22, 8 (Aug. 1973), 710-717.

[Mins67] Minsky, M. L. Computation: Finite and Infinite Machines, Prentice-Hall, Englewood Cliffs, N. J., 1967.

[Naur60] Naur, P. Ed. Report on the algorithmic language ALGOL 60. Comm. ACM 3, 5 (May 1960), 299-314.

[Owic75] Owici, S. Axiomatic proof techniques for parallel programs. Ph.D. Th., Cornell U., 1975.

[Parn72] Parnas, D. L. On the criteria to be used in decomposing systems into modules. Comm. ACM 19, 12 (Dec. 1972), 1053-1058.

[Pati71] Patil, S. S. Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes. Comput. Structures Group Memo 57, Project MAC, M.I.T., Cambridge, Mass. Feb. 1971.

[Pete74a] Peterson, J. L. Modelling of parallel systems. Ph.D. Th., Dept. of Electr. Eng., Stanford U., 1974.

[Pete74b] Peterson, J. L., and Bredt, T. H. A comparison of models of parallel computation. Proc. IFIP Cong. 74, North-Holland Pub. Co., Amsterdam, 1974, pp. 466-470.

[Petr66] Petri, C. A. Communication with automata. Suppl. 1 to Tech. Rep. RAD C-TR-65-337, vol. 1, Griffiss Air Force Base, New York, N. Y., 1966. (Translated from Kommunikation mit automatin, U. Bonn, Bonn, Germany, 1962.)

[Ridd72] Riddle, W. E. Modelling and analysis of supervisory systems. Ph.D. Th. and Tech. Rep. STAN-CS-72-271, Computer Sci. Dep., Stanford U., Stanford, Calif., March 1972.

[Ridd73] Riddle, W. E. A design methodology for complex software systems. Proc. 2nd Texas Conf. on Computing Systems, 1973, pp. 22.1-22.8.

[Ridd76] Riddle, W. E. An approach to software system modelling, behavior specification and analysis. RSSM/25, Dept. of Comptr. and Comm. Sci., U. of Michigan, Ann Arbor, Mich., July 1976.

[Ridd77a] Riddle, W. E. et al. An introduction to the DREAM software design system. Software Engineering Notes 2, 4 (July 1977).

[Ridd77b] Riddle, W. E. A formalism for the comparison of software analysis techniques. RSSM/29, Dept. of Comptr. and Comm. Sci., U. of Michigan, Ann Arbor, Mich., July 1977.

[Rutl64] Rutledge, J. D. On Ianov's Program Schemata. J. ACM 11, 1 (Jan. 1964), 1-9.

[Sang77] Sanguinetti, J. W. Performance prediction in an operating system design methodology. Ph.D. Th., Dept. of Comptr and Comm. Sci., U. of Michigan, Ann Arbor, Mich., May 1977.

[Simo62] Simon, H. A. The architecture of complexity. Proc. of the Amer. Philo. Soc. 106, 6 (Dec. 1962), 467-482.

[Turi36] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. Proc. London Math. Soc., Ser. 2-42, 230-265.

[Welt76] Welter, M. F. Counter expressions. RSSM/24, Dept. of Comptr. and Comm. Sci., U. of Michigan, Ann Arbor, Mich., Oct. 1976.

[Wile76] Wileden, J. C. Derivatives of message transfer expressions. RSSM/22, Dept. of Comptr. and Comm. Sci., U. of Michigan, Ann Arbor, Mich., Feb. 1976.

[Wirt71] Wirth, N. Program development by stepwise refinement. Comm. ACM 14, 4 (Apr. 1971), 221-227.

[Wulf74] Wulf, W., Levin, R., and Pierson, C. Overview of the HYDRA operating system development. Proc. Fifth Symp. on Operating Systems Principles, Operating Syst. Rev. (ACM SIGOPS Newsletter) 9, 5 (Nov. 1975), 122-131.

[Zave77] Zave, P., and Fitzwater, D. R. Specification of asynchronous interactions using primitive functions. Dept. of Comp. Sci., U. of Wisconsin, Madison, Wisc., 1977.

[Zerv77] Zervos, C. R. Colored Petri nets: Their properties and applications. Ph.D. Th. and SEL Tech. Rep. 107, Dept. of Electr. and Comptr. Eng., U. of Michigan, Jan. 1977.