A SOFTWARE STRUCTURING TOOL FOR MESSAGE-BASED SYSTEMS

by

KIM LAWSON ROCHAT

B.S., Kansas State University, 1975

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1980

Approved by:

Major Professor

THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH THE ORIGINAL
PRINTING BEING
SKEWED
DIFFERENTLY FROM
THE TOP OF THE
PAGE TO THE
BOTTOM.

THIS IS AS RECEIVED
FROM THE
CUSTOMER.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

ACKNOWLEDGEMENTS

This research was supported in part by the Army Institute for Research in Management, Information, and Computer Systems under grant number DAAG 29-78-G-0200 from the Army Research Office.

I would like to thank Robert Young for providing the software necessary to support the development of S^3T, as well as the other members of the OS Group, Roxanna Fundis and Rich Sanders, for providing valuable input for the design decisions made concerning S^3T.

I would especially like to thank Dr. Wallentine for the faith he has expressed in me and the guidance which he has provided over the last eight years.

1. INTRODUCTION

1.1 OVERVIEW

The Software Systems Structuring Tool (S^3T) is a tool which assists in the definition and construction of software configurations which are executed by message-based systems. This tool supports the typed interconnection of modules which allows the verification of the correctness and completeness of interconnection, incremental construction of configurations, and an implementation independent structure representation. S^3T uses independently compiled modules with typed ports to construct distribution-independent configurations.

1.2 TERMINOLOGY

A message-based system is a collection of active entities, usually processes, communicating with each other by sending messages on channels. The active entities of a message-based system will be termed modules. Most message-based systems consider modules to be processes, but this reflects both an implementation and operating systems view. The term module has been selected for its software engineering connotation of an encapsulated functional unit.

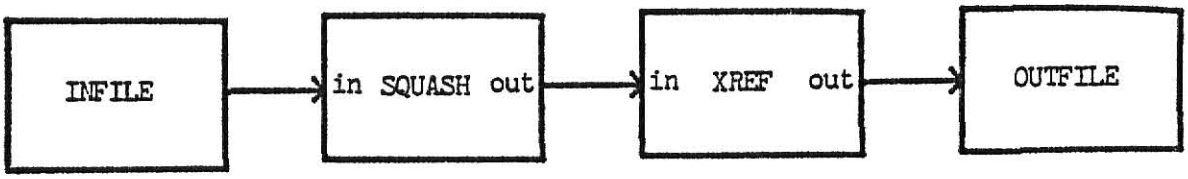
The point at which a module is attached to a channel is a port [2]. Each module in a configuration has at least one port. Not every message-based system has ports. Many language-based systems requiring single compilation do not have ports. Instead they use process names

and variable names within these processes to indicate which data designator in one process is to be connected to a data designator in another process.

A channel is a simplex transmission stream. Bi-directional communication between two processes requires two channels. A channel is an object in a message-based system to which one or more modules or ports may be attached. Channels reflect an implementation view of a message-based system. Language-based systems typically contain a connect operation on ports. The result of a connect operation is a connection.

The module or port at the end of the connection nearest the user (or the one currently being described) is referred to as the local module or port. The connecting module or port is remote.

A message-based system may be considered to be a directed graph with the active entities forming the nodes and the channels or connections forming the arcs. A particular system graph is a software configuration.



The software configuration shown has four nodes. There are two nodes which execute programs, each node having ports named IN and OUT. The other two nodes execute system routines which access files. The SQUASH program replaces two adjacent asterisks in the input with a single caret in the output. The XREF program cross-references program source files.

Most systems contain initialization parameters which are constant

parameters to a module when it is initialized. These parameters may be channel identifiers [19], program parameters [24], the name of the program to execute, or some other attribute to invoke a particular instance of the module.

1.3 SURVEY OF MESSAGE-BASED SYSTEMS

Message-based systems may be divided up into two groups, operating systems with their command languages and language-based systems. The operating systems primarily support channels and processes as resources which may be configured together at execution time. The language-based systems are intended to be software engineering tools, i.e. providing a "better" or "more natural" solution to certain programming problems.

All of the message-based systems considered here may be configured or reconfigured in software, i.e. requiring no hardware changes. Some systems are static, and must be recompiled to reconfigure them [6,12]. Others may be configured at initiation of execution [5], and some can be reconfigured during execution [13].

Typically language-based systems have strongly typed communications, but are static due to compile-time type checking. The operating systems are untyped and dynamic.

The characteristics of some message-based systems of both types are summarized in Table 1. This table is adapted from Hunt [13] and is extended to include the NADEX operating system [24] and S^3T.

The systems surveyed are listed in Table 1 chronologically by system type. Operating systems which support message-based interprocess communication are listed with a system type of OS. Command languages which allow a user to describe a software configuration at command time

are listed as system type CL. Programming languages and systems which support configurations either primarily, such as CSP [12], or incidentally, such as CPascal [6], are listed with the system type PL. The Software Systems Structuring Tool is listed at the bottom of Table 1 for comparative purposes and will be discussed later.

The column headed connect type lists the type of object in the structuring system, if any, which is typed. Typed channels mean that the channel connecting two modules will only pass messages of one particular type. Typed messages (Msg) means that two modules may communicate only if messages of the same type are passed between them. Normally only one type of message is permitted per intermodule connection, but CSP [12] allows two modules to synchronize on arbitrary message types at execution time. Here, as throughout the remainder of Table 1, NA means "not applicable"

The column headed ports indicates whether each system encapsulates the module connection interface so that modules can be connected independently at a time following module specification. Note that all of the operating systems have ports, but not all of the language systems. This is so the language systems can support compile-time typing of module connections.

The column headed buffered indicates whether the facility supporting the message interface allows more than one message in a channel.

The column headed dynamic config indicates whether the system allows configuration at invocation time. Again, the systems which do not allow dynamic configuration are those language systems which are compiled, and DSLM [17] which requires the assembly of macros before

execution.

The column headed dynamic reconfig indicates whether a configuration can be restructured during execution. This restructuring may mean either rearranging or creating new connections, or creating new modules and connecting them to existing modules.

The columns headed per channel readers writers refer to the number of ports (or modules) which may be attached to each end of a channel.

The columns headed per module reads writes refer to the number of outstanding operations a module may have on all of its ports.

The column headed multiplexor describes who is responsible for scheduling the acceptance of messages in those systems which support non-determinacy. Every system which allows many writers per channel implements message (msg) scheduling. The messages are received in the order sent, and the user has no control over which message is received. If the module allows more than one read to be outstanding, then either the User or the Implementation (Impl) determine which message is to be processed next. The NADEX system is an interesting example of these scheduling techniques. If the user wishes to provide his own scheduling mechanism, then he writes the module in Sequential Pascal and uses the system primitive AWAIT_EVENTS to notify him of which "reads" are ready. He can then take his choice of the available messages or wait for another event. If the user does not want to explicitly schedule "reads", the module can be written in Concurrent Pascal with a process serving each port. The kernel will then schedule the execution of processes as messages arrive. The scheduling of messages is discussed in further detail in Appendix 1.

name	system type	connect type	ports	buffered	dynamic config	dynamic reconfig	per channel readers	per channel writers	per module reads	per module writes	multi-plexor
Balzer[2]	OS	NA	Yes	Yes	Yes	Yes	1	1	Many	Many	User
Walden[22]	OS	NA	Yes	No(9)	Yes	Yes	1(2)	Many(5)	Many	Many	User
DEMOS[3]	OS	Channel	Yes	Yes	Yes	Yes	1	Many	Many	1	User
UNIX[19]	OS	NA	Yes	Yes	Yes	No	Many	Many	1	1	Msg
DSLM [17]	OS	NA	Yes	Yes	No	No	1	Many	1	1	Msg
StarOS[14]	OS	Msg	Yes	Yes	Yes	Yes	Many	Many	Many	Many	User
NADEX[24]	OS	NA	Yes	Yes	Yes	Yes(10)	1	Many	Many	1	(11)
UNIX Shell[5]	CL	NA	Yes(12)	NA	Yes	No	1	1	NA	NA	NA
MIRACLE[10]	CL	NA	Yes	NA	Yes	Yes	Many	Many	NA	NA	NA
CPascal[6]	PL	Msg	No	Yes	No	No	Many	Many	1	1	Impl
Mesa[11]	PL	Msg	Yes	No	Yes	Yes	1	Many	1	1	User
CSP[12]	PL	Msg(1)	No	No	No	No	1(4)	1(4)	Many(7)	1(8)	Impl
LIMP[13]	PL	Channel	Yes	No	Yes	Yes	Many	Many	1(6)	1(6)	Msg
C-UCSD[1]	PL	NA	Yes	Yes	Yes	Yes	1	Many	1	1	Msg
S^3T	Tool	Port	Yes	NA	Yes	No	1	1	NA	NA	NA

Notes:
1. Run-time pattern matching.
2. Although the potential for multiple readers and writers exists.
3. Each reader gets the same input sequence.
4. But the processes may be subdivided to give several communicants.
5. Due to RECEIVE ANY.
6. Can program multiple reads and writes by using an extra process.
7. Uses guarded commands.
8. Contemplated.
9. Buffers must be provided by user.
10. Planned.
11. Depends on language - User for Spascal, Impl for Cpascal.
12. Ports are implicit.

Table 1. Summary of message-based systems

The surveyed systems will be discussed in turn, with a particular emphasis on the structuring techniques each uses to describe software configurations.

Balzer [2] introduced ports as a system structuring technique. Two ports are connected by each having a pointer to the other. A system monitor is proposed to service the connection requests of each module.

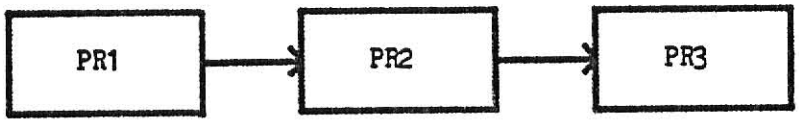
Walden [22] proposes a system in which connections exist only for the duration of a single message. All ports in the system are uniquely named. Each send or receive request must name the corresponding port. If two processes wish to communicate, the system monitor is requested, through a well-known port, to allocate a pair of port numbers to be used for the duration of a "connection" (in the normal sense of many messages).

DEMOS [3] supports communication between processes across protected objects called links. Links provide message paths between processes. These objects (links) can be passed among processes across other links. Thus dynamic reconfiguration at execution time is possible.

UNIX [19] allows processes and pipes to be created dynamically. Configurations are structured by a parent process who allocates pipes for communications and forks processes with the appropriate pipe identifiers as start parameters. For instance, a pipeline with three processes connected by two pipes could be created as follows:

```
PIPE1 := PIPE();
PIPE2 := PIPE();
PR1 := FORK(PIPE1);
PR2 := FORK(PIPE1,PIPE2);
PR3 := FORK(PIPE2);
```

The software configuration created looks like:



DSLM [17,16] is a data-driven transaction-based system. Systems are configured using assembly language macros.

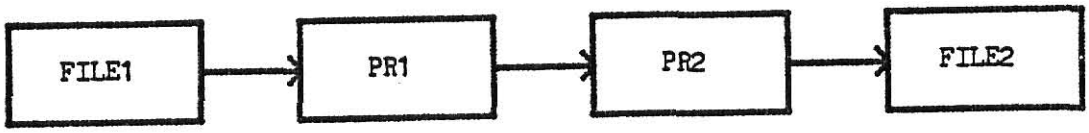
StarOS [14] is a capability-based operating system which permits dynamic creation of capability-based objects which include pipes. These can be used by the programmer to dynamically construct a graph of program nodes at execution time whose arcs consist of pipes.

NADEX is a "portable" (written in Concurrent Pascal) operating system. Configurations are structured by a Configuration Descriptor record which describes the modules and their port connections. The configuration descriptor is submitted to the operating system by the command processor for execution.

The UNIX Shell [5] implements a syntax for dynamically (at command time) configuring pipelines within the UNIX operating system. The command

PR1<FILE1 | PR2 | PR3 >FILE2

describes the linear configuration



Module ports are implicitly named.

MIRACLE [10] is a proposed network command language. It allows the dynamic construction of arbitrary configurations. The syntax for

the configuration illustrated above is:

```
PR1 PORT1<FILE1 PORT2>*1 ||
PR2 PORT1<*1 PORT2>*2 || PR3 PORT1<*2 PORT2>FILE2
```

where *<integer> is a channel identifier. The portnames are explicitly connected to channels. These connections may be simplex in either direction ('<','>') or duplex ('<>'); No specific underlying implementation is proposed.

Concurrent Pascal [6] is a monitor-based language, however any data flow between processes must be in the form of messages with monitors forming the channels. Several CPascal programs resemble configurations, including the SOLO operating system [6]. Configurations are static at compile time. If CHANNEL1 and CHANNEL2 are monitors implementing a bounded message buffer and PR1, PR2, and PR3 are processes, then at compile-time the configuration of the pipeline would be:

```
INIT CHANNEL1, CHANNEL2,
     PR1(CHANNEL1),
     PR2(CHANNEL1, CHANNEL2),
     PR3(CHANNEL2);
```

Mesa [11] allows a process to declare instances of modules having ports and connecting them:

```
JOIN PR1.PORT1 TO PR2.PORT1;
JOIN PR2.PORT2 TO PR3.PORT1.
```

Ports are typed by the type of message transmitted across them. The message type must be compatible before a JOIN operation can be performed. Channels are called links and are simplex. A link may be passed between modules so that configurations may be dynamically restructured.

Communicating Sequential Processes [12] is a proposal for a concurrent systems language. Processes must explicitly name their

communicant process at compile time, so configurations are static and modules cannot be independently compiled. Messages are dynamically typed so that a RECEIVE for a particular message type will wait for a SEND of the corresponding message type. This means that messages will not necessarily be received in the order in which they are sent. If MSG is a message variable, then

```
[PR1::PR2!MSG || PR2::PR1?MSG; PR3!MSG || PR3::PR2?MSG]
```

represents the pipeline configuration.

The Language for Implementing Messages and Processes (LIMP) [13] has message typed channels and capabilities for ports to perform READ and WRITE operations on a particular channel type. Port capabilities and channels may be passed freely between modules, allowing dynamic reconfiguration.

Concurrent UCSD Pascal is a set of language extensions to UCSD Pascal to allow message-based systems. Processes may be dynamically created. Associated with each process is a channel with a unique identifier. Other processes having access to this identifier may write to the process's channel. Configurations must be initialized by the process who created the processes. The channel identifiers of communicating processes must be written to the remote process by the configuring process.

1.4 MOTIVATION FOR S^3T

The Software Systems Structuring Tool is an attempt to integrate the common aspects of the massage-based systems into a software engineering tool for the construction of configurations. This tool is implementation-independent, supports strongly typed interconnection of

modules, and allows the construction of incomplete configurations which may be encapsulated into modules in a hierarchical manner. S^3T uses independently compiled modules with strongly typed interfaces using ports to construct configurations.

Other module structuring tools in the literature for assembling software systems include the Module Interconnection Language [8] and the Module Control System [18]. Module Interconnection Language is for constructing hierarchies of procedures. The contribution of [18] is the concept of a separate typed module interconnection language. The Module Control System uses this concept for structuring the system components of a CPascal program in a hierarchy.

1.5 ORGANIZATION OF PAPER

Chapter two contains a discussion of module types and their encapsulation. The type-safe interconnection of modules to form configurations and the encapsulation of configurations are discussed in Chapter 3. The implementation of the Software Systems Structuring Tool in conjunction with the NADEX operating system is presented in Chapter 4. Chapter 5 contains a discussion of some of the experiences with S^3T, a discussion of portability, and proposals for further work. Appendix 1 contains an example of structuring the dining philosophers problem [12] for several different implementations. Appendix 2 contains three example user sessions illustrating the incremental construction of a software configuration.

2. MODULE CONSTRUCTION

2.1 MODULE TYPES

In general, there are two types of modules - those which produce or consume data, and those which tranform data. A data producer or consumer module may encapsulate a file, a device, or another data source or sink. A source or sink can be modeled (and in fact is implemented in S^3T) as a module with one port. A module which transforms data is a program module. Program modules are constructed by users of the system and kept available in a module library. Every program module has associated with it a record which defines the resource requirements for the execution of the module and the interface between the module and the user (or the other modules in the configuration). This record is called a Partial Configuration Descriptor (PCD) record for reasons which will be seen in the next chapter. The PCD record corresponds to the definitions module of Mesa[11] or the resource attribute descriptor of Gray [10].

2.2 PROGRAM MODULES

Module programming, or "programming in the small" [8] is the activity of coding, testing, and encapsulating a module which will later be included in a configuration.

Operating systems for message-based systems usually do not provide any special capabilities to support module programming. The module

programming language may be the operating system language; assembler [17], C [5], or Pascal [24]. The operating system module languages are usually extended to implement ports.

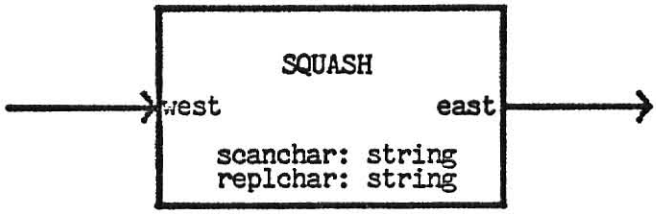
2.3 INTERFACE SPECIFICATION

The interface to a module consists of a unique module name, a list of the ports and their types, and a list of the module initialization parameters with their types and default values. Given this interface, the implementation details of the module can be hidden from the user. The interface specification also contains the resource requirements of the module so that it can be successfully invoked by the operating system.

The program shown in Figure 1a is written in Sequential Pascal extended with ports. This program is a generalization of the SQUASH program given in [12]. Its function is to replace two adjacent SCANCHAR characters in the input text with a single REPLCHAR character in the output text. The program has two ports which access ASCII characters. The input port is port one. It is referred to using the constant integer WEST. The output port is port two. It is referred to using the constant integer EAST. The implementation demonstrates knowledge of the ASCII transfer protocol in its use of the End_Medium character (EM) to determine end of stream.

```
PROGRAM squash(scanchar, replchar: CHAR);      "******************"
  CONST west = 1;  east = 2;                   "*  S Q U A S H  *"
  VAR c: CHAR;                                 "******************"
BEGIN
  REPEAT
    READ_CHAR(west,c);
    IF c <> scanchar THEN WRITE_CHAR(east,c)
    ELSE BEGIN
         READ_CHAR(west,c);
         IF c = scanchar THEN WRITE_CHAR(east,replchar)
         ELSE BEGIN WRITE_CHAR(east,scanchar); WRITE_CHAR(east,c); END;
         END;
  UNTIL (c = em);
END.
```

a. SQUASH program listing.



b. Conceptual encapsulation of SQUASH program module.

```
TYPE squash_program = PROGRAM(scanchar: STRING;
                              replchar: STRING);

  PORTS
  1 west     : MODE = READ, PROTOCOL = ASCII, RECORD_LENGTH = 1;
               MINIMUM DATA BUFFERS = 1, MINIMUM PARAMETER BUFFERS = Ø;
  2 east     : MODE = WRITE, PROTOCOL = ASCII, RECORD_LENGTH = 1;
               MINIMUM DATA BUFFERS = 1, MINIMUM PARAMETER BUFFERS = Ø;

  ATTRIBUTES
    PREFIX = NATIVE;
    CODE_FILE = sys2:squash.img/p;
    CODE_SPACE = 1K;
    DATA_SPACE = 1K;
    OVERLAY_SPACE = ØK;
    PARAMETERS = scanchar: STRING;
                 replchar: STRING;

END PROGRAM;
```

c. PCD describing encapsulation of SQUASH program.

Figure 1. SQUASH program and its encapsulation.

Figure 1b is an illustration of the conceptual encapsulation of the SQUASH program. The module name, parameter names and types, as well as the port names and data flow directions are visible.

The program must be included in a configuration before it can used. There are three ways in which this might be accomplished. The author of the program could create a static configuration which accepts parameters for the filenames to read and write as well as the program parameters. This is the inflexible approach taken by traditional operating systems.

The ports could be assumed to be standard input and output ports as in the UNIX system. The module could then be included in a linear configuration by a casual user.

```
SQUASH <INFILE  >OUTFILE
```

See, it's easy! What? Oh, the program parameters! Why doesn't anyone remind me?

The user has forgotten to specify the parameter values for SCANCHAR and REPLACECHAR. In the real UNIX system, the program alone is responsible for parsing its parameters and providing help messages. If the command processor parses command parameters and passes them to the program, then

```
SQUASH '*' '^' <INFILE >OUTFILE
```

will result in a successful invocation.

There are several other problems with UNIX-style configuration construction. Configurations containing coroutining modules, or for that matter, any other non-linear configuration cannot be constructed without bidirectional communications using arbitrary message types and arbitrary configurations. The construction of arbitrary configurations

requires the user to be aware of the module port names and types as well as the initialization parameters.

Under the S^3T system, a user can type in the name of a module:

SQUASH

and receive adequate interface information to allow him to use it:

```
UNCONNECTED PORTS - TRY AGAIN:
    squash  [scanchar: STRING DEFAULT('*')],
            [replchar: STRING DEFAULT('^')]
            PORTS:
                  in : ASCII READ,
                  out: ASCII WRITE
```

This interface information indicates that there are two program parameters named SCANCHAR and REPLCHAR of type string (really character). The values '*' and '^' will be supplied for the parameters if the user does not specify them. The module has two ports named IN and OUT. IN expects to READ ASCII characters, OUT expects to WRITE ASCII characters. Using this information, the user can confidently include this module in his configuration:

```
SQUASH '.' ':' IN<INFILE OUT>*1 || XREF 'IM,SU' IN<*1 OUT>OUTFILE
```

This command was written by a person who prefers his Pascal program listings with PL/1 range specifiers (':') instead of Pascal's "up_to" ('..'). The port interface information can also be used to type-check the ports specified in a connect operation. This aspect is discussed in detail in the next chapter.

There is still an interface problem - the user may *not* have the foggiest notion of the function of the module. Hopefully well chosen module, parameter, and port names will provide clues. Optimally, a paragraph of text would be included in the interface specification to describe the function and use of the module. However, some modules

(such as the File Subsystem of NADEX [24]) require a report to fully explain their use.

Figure 1c contains an illustration of the contents of the module interface definition record used by S^3T in text form. The illustrated interface is taken from the NADEX implementation of S^3T but simplified to remove implementation dependent resource information.

The module interface gives the name of the module, specifies that it encapsulates a *program* (rather than a partial configuration, which will be explained in the next chapter), and lists the module parameters with their types and default values.

The types supported by S^3T are boolean, integer, 32 character string, eight character identifier, implementation-dependent file descriptor, and implementation-independent pathname.

Each module parameter may have a default value supplied for it. The most recently supplied value is used for the parameter value at invocation time. If no other value is specified, then the default value is used. Program module parameters may either be passed to the program when it is invoked or used to supply resource information. For instance, if the amount of data space a program requires is a function of its initialization parameters, the data space for the program could be provided by a parameter. The current implementation of S^3T always references parameters by value and does not allow multiple references to a parameter.

Next, the ports of the program are described. The port number is used to correlate between the port number accessed by the program and the port declaration in the interface specification. The port name is next. All port names within a module must be unique, so that any port

in a configuration can be accessed using the module instance name and the portname. The type of the port is its protocol. Port protocols are discussed in depth in the following chapter. In the current implementation of S^3T, the port protocol consists of three attributes; MODE, PROTOCOL, and RECORD_LENGTH. These attributes are derived from those supported by the NADEX File Subsystem[24]. MODE describes the direction of information flow through the port. Possible modes are READ, WRITE, or READ_WRITE. The PROTOCOL attribute describes agreed upon sequences of operations for commonly used functions. SEQUENTIAL and ASCII protocols are used to access files or other program modules. RANDOM protocol is a request-response protocol currently only used for files, although it is similar to a coroutine protocol. For ports having coroutine protocols, i.e. primarily exchanging control information rather than data, a USER protocol type may be provided. An identifier supplied by the user provides a unique protocol name for those ports intended to use the services of a particular server module. RECORD_LENGTH is an attempt to provide typed messages by length, similarly to the UNIV parameter type declarations of Sequential Pascal[6].

The next section of the program module description is the program ATTRIBUTES. These provide resource information about the program to the operating system. This information includes a pointer to the program code file, the memory requirements necessary for execution, and the names and order of the initialization parameters which are to be passed to the program at its invocation.

S^3T maintains the module interface information illustrated in Figure 1c in a partial configuration descriptor (PCD) record. After

program modules have been described, they may be interconnected to create partial or complete software configurations.

3. CONFIGURATION CONSTRUCTION

3.1 MODULE INTERCONNECTION

Modules are interconnected to form software configurations through the use of the CONNECT operation on module ports. Only two ports may be connected. Although many of the systems support attaching several ports to a channel, this will not be considered since it is not the most general case and it is not clear that it is a good structuring technique for software configurations. Only the creation of static configurations which cannot be reconfigured after invocation is considered.

3.2 PORT COMPATIBILITY

If ports are considered to be objects upon which operations can be performed [2], then software engineering practice suggests that the operations on ports--connect, disconnect, read, and write--be typed. This typing may occur in several ways. It could be as simple as checking that the operands of a port operation are indeed ports. Recent work [11,12,13,14] has suggested that connect operations be typed by the data object passing through the ports. CSP[12] allows dynamic typing with the result that several different message types may be passed through the same port and selectively received. The Software Systems Structuring Tool extends these concepts to include typing ports being connected by the protocol of the ports being connected. Protocol includes, in addition to the type of mesages being transferred, the set

of operations which may be performed on the port (i.e. READ and WRITE), and the sequence of port operations.

The following discussion concerning typed ports is given from a software engineering standpoint. The techniques suggested are inefficient in many message-based systems, since they range from sending multiple messages of different types instead of one message to the run-time validation of port operations against a formally specified protocol. The discussion is given with the idea that type-safe module interconnection is a desirable goal and that some thought about the problem should be stimulated.

A reasonable goal would be compile and connection time type checking of as many aspects of the protocol as possible. Ideally the compiler could generate enough information about operations on two ports to guarantee that they will communicate correctly.

The port can be restricted to pass messages of only a single message type or allowed to pass a variety of message types. If a port is restricted to a single type, its message type compatibility can be determined by the compiler, either by message length, or a global typing system such as found in [11].

Typically, a port passing only a single message type packs fields for all possible request types, their parameters, and responses into the message type record. Some type security is guaranteed if the record is a variant record [6], but the parameter modes of the individual fields cannot be assured. For instance, a parameter to a certain request type (constant mode) may be inadvertently omitted, or a parameter may not be returned by the server module as expected (var mode).

The obvious way to guarantee the parameter modes is to pass each

constant parameter as a separate message and then read each response parameter in sequence. This causes the simple WRITE; READ coroutine request-response protocol to become a complicated sequence of writes and reads of various message types.

The port mode is the direction in which data passes through the port. Mode can also be considered to represent the set of allowable operations on the port. There are three general modes: READ, WRITE, and READ_WRITE. Ports which are READ only or WRITE only typically appear in data flow driven configurations. Coroutine modules will use READ_WRITE mode for their request-response protocol. Therefore port compatibilty checking by mode and record type might be suitable for a data driven configuration, it will not be rigorous enough for coroutines. One way in which port modes could be tested for compatibility is shown in Table 2. This guarantees that two ports whose message types are the same but both expect to read cannot be connected.

Only one current system[13] will prevent a connection from being made between two ports who both intend to WRITE (or READ) the same message type.

mode	R	W	RW
R	No	Yes	Yes
W	Yes	No	Yes
RW	Yes	Yes	Yes

Table 2. Port mode compatibility

As an initial attempt at defining typed ports for coroutine protocols, the user could provide a unique protocol name for the port type. Only those ports having the same protocol name could be interconnected. However, this approach assumes that the module follows the named protocol and does not prevent the connection of two ports who both

expect to use the services of the remote server module.

The next refinement is the creation of a library of compatible user protocol names. For example, the NADEX File Subsystem [24] is a server module which provides file services and uses a unique protocol. The File Subsystem provides services for several users. The File Subsystem ports could be given a protocol type name of 'FSS_SERVICE'. The modules wishing to use the File Subsystem would have protocol names of 'FSS_REQUEST'. In the library would be a protocol compatibility entry of (FSS_SERVICE, FSS_REQUEST) indicating that a port of type FSS_SERVICE may be connected to a port of type FSS_REQUEST. Notice that two ports of the same type, either FSS_SERVICE or FSS_REQUEST cannot be connected.

Finally, an approach porposed in this thesis is for user-supplied protocol information to use path expressions [7] to describe the order and type of operations performed on a port. The objective is to permit the programmer of a module to specify, in the language, the valid port transitions. The compiler can then check for valid port operations and types as well as generate code for run-time checks for valid module state transitions when operating on ports. Thus the protocol violations of a module interface (port) can be checked at run-time.

For example, a port type declaration for a user of the NADEX File Subsystem might be:

```
TYPE page = ARRAY[page_indx] OF BYTE;

TYPE fsr_types = (fsr_assign, fsr_close, fsr_alloc, fsr_delete,
                  fsr_lookup, fsr_response)

TYPE fsr_codes = (fsr_ok, fsr_error);

TYPE file_system request = RECORD
                             fsr_type: fsr_types;
                             fsr_result: fsr_codes;
                             "other fields for function codes"
                           END;

TYPE fss_port = PORT file_system_request.fsr_type,
                     file_system request.fsr_result,
                     page

PROTOCOL

   ( WRITE(fsr_assign);
       ( ( READ(fsr_ok);
             ( {READ(page)} , {WRITE(page)} );
              WRITE(fsr_close); READ(fsr_result)
         ),
           READ(fsr_error)
       )
   ),

   ( WRITE(fsr_close); READ(fss_result) ),

   ( WRITE(fsr_delete); READ(fss_result) ),

   ( WRITE(fsr_alloc); READ(fss_result) ),

   ( WRITE(fsr_lookup);   READ(fss_result) )  END;
```

The path expression enclosed by PROTOCOL ... END may be repeated any number of times. The READ and WRITE operations specify the type of message passed as a parameter.

Since the transition between port states may depend on a value, such as whether or not the response to a request is "ok", an enumeration constant may be a parameter of a port operation. For the purposes of this example, it is assumed that all enumeration constants are unique, so that their defining enumeration type can be identified. The types listed following the PORT keyword enumerate the types of messages which

may flow through the port. Any enumeration type appearing in this list may be used to drive port transitions. In the example above, the inclusion of FILE_SYSTEM_REQUEST.FSR_TYPE in the port type list indicates that messages of type FILE_SYSTEM_REQUEST may be passed throught the port, and the value of the FSR_TYPE enumeration field can be used to select port transitions.

Semicolons sequence port operations. Operations in braces '{ }' may be repeated one or more times (repetition). A list of operations separated by a comma indicates that any one of the operations may be chosen (alternation). An algorithm would then be used to compare two path expressions for compatibility when they are connected.

The port protocol illustrated above is for a user of the NADEX File Subsystem. There are five file system functions which may be called in any order. These are ASSIGN, CLOSE, DELETE, ALLOCATE, and LOOKUP. All functions except ASSIGN are simple request-response protocols. The type of request written determines which transition the port makes initially. If the response from an ASSIGN is fsr_error (for instance, the file is protected or cannot be found), then the ASSIGN path is terminated. Otherwise one or more READ operations or one or more WRITE operations are performed, followed by a CLOSE request and its response. Although this is more rigorous typing than that found in any existing system, there is at least one problem with this approach.

This problem concerns deciding when to make the transition out of the loop READING or WRITING pages. NADEX uses an out of band signal to indicate the end of data transmission. The port path must be able to detect this transition, either by the addition of a field to a page which indicates the next transition, or by the receipt of a different

type record marking End_of_File.

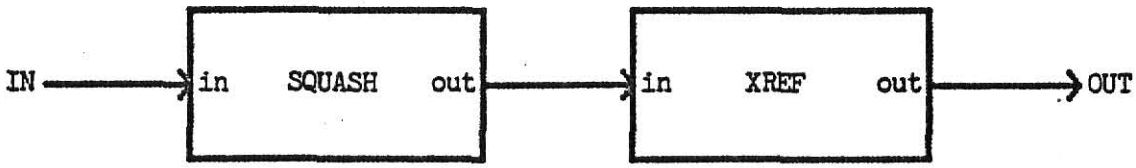
Any of the port typing techniques which require the formal specification of protocol have the additional advantage of documenting the interconnection protocols of ports.

To save the user the necessity of specifying port types, a technique of importing type definitions such as that used in [11] could be used to build a library of port types. Whenever a particular port type is required, it could be obtained from the library. Appendix 1 contains an example which uses port path expressions and a type definitions module.

3.3 CONFIGURATION ENCAPSULATION AND EXTERNAL PORTS

Figure 2 contains an illustration of how a module can be defined which encapsulates a portion of a configuration. Modules can be declared as local constants, or NODES. The ports of local modules are designated using the construct <nodename.portname> . The ports of the included modules must either be CONNECTed to another port within the module description or exported, or made visible, outside of the module by designating it as an EXTERNAL PORT. At each level of encapsulation (within each module description) the exported port names may be renamed. The INIT statement passes constant and parameter values to the subcomponents to instantiate this invocation. The ports of a CONNECT operation are checked for compatibility by S^3T.

Appendix 2 contains the sample sessions with the Interactive PCD Constructor which actually created the module specifications shown in Figures 2 and 3.



a. Conceptual ecapsulation of SQUASH and XREF modules.

```
TYPE sqx = MODULE;

  NODES
    sq          : squash;
    x           : xref;

  INIT
    sq('.', ':');
    x('IM,SU');

  EXTERNAL PORTS
     in         : sq.in;
     out        : x.out;

  CONNECT
    sq.out TO x.in;

END MODULE;
```

b. PCD describing the partial configuration.

Figure 2. Ecapsulation of a partial configuration.

a. Conceptual encapsulation of the complete configuration.

```
TYPE sqxref = module(input   : FD;
                     output  : FD);

  NODES
    sqxrf    : sqx;
    infile   : FILEACCESS;
    outfile  : FILEACCESS;

  INIT
    sqxrf;
    infile(FILENAME = input, MODE = READ,
           PROTOCOL = ASCII, RECORD_LENGTH = 1);
    outfile(FILENAME = output, MODE = WRITE,
            PROTOCOL = ASCII, RECORD_LENGTH = 512);

  CONNECT
    sqxrf.in TO infile.;
    sqxrf.out TO output.;

END MODULE;
```

b. PCD describing the complete configuration.

Figure 3. Description of a complete configuration.

A software configuration must be completely connected, i.e. have no unconnected ports, before it can be executed. With Figure 3, the partial configuration of Figure 2 is incorporated into a complete configuration by adding file access nodes to the configuration. The names of the input and output files are supplied by the module parameters INPUT and OUTPUT.

It has now been shown how a software configuration can be constructed incrementally using S^3T. The next chapter contains a description of the implementation of S^3T in the NADEX environment.

4. THE SOFTWARE SYSTEMS STRUCTURING TOOL

4.1 DESCRIPTION OF S^3T

The Software Systems Structuring Tool (S^3T) is a system for constructing software configurations which has been implemented as part of the job control system of the NADEX operating system[24]. S^3T is an implementation-independent system for the definition software configurations [2]. S^3T consists of a library of files and three programs which operate upon them. Each file contains a record which symbolically describes a complete or incomplete software configuration. These records are termed PCD records, an abbreviation for Partial Configuration Descriptor. These descriptors are similar to the Resource Attribute Descriptor of Gray [10] or the definitions module of Mesa [11]. Each descriptor describes the resources needed by and the services provided by the module. If this PCD describes a program, then its language type, memory requirements, and parameters are described.

A PCD describes the services provided by a module in the form of a list of ports and a list of parameters. PCD records may encapsulate other PCD records. For additional information concerning the implementation of S^3T, see [20].

4.2 IMPLEMENTATION

The three programs which operate upon PCD files are the Interactive PCD Constructor Program, the PCD De-compiler Program, and

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

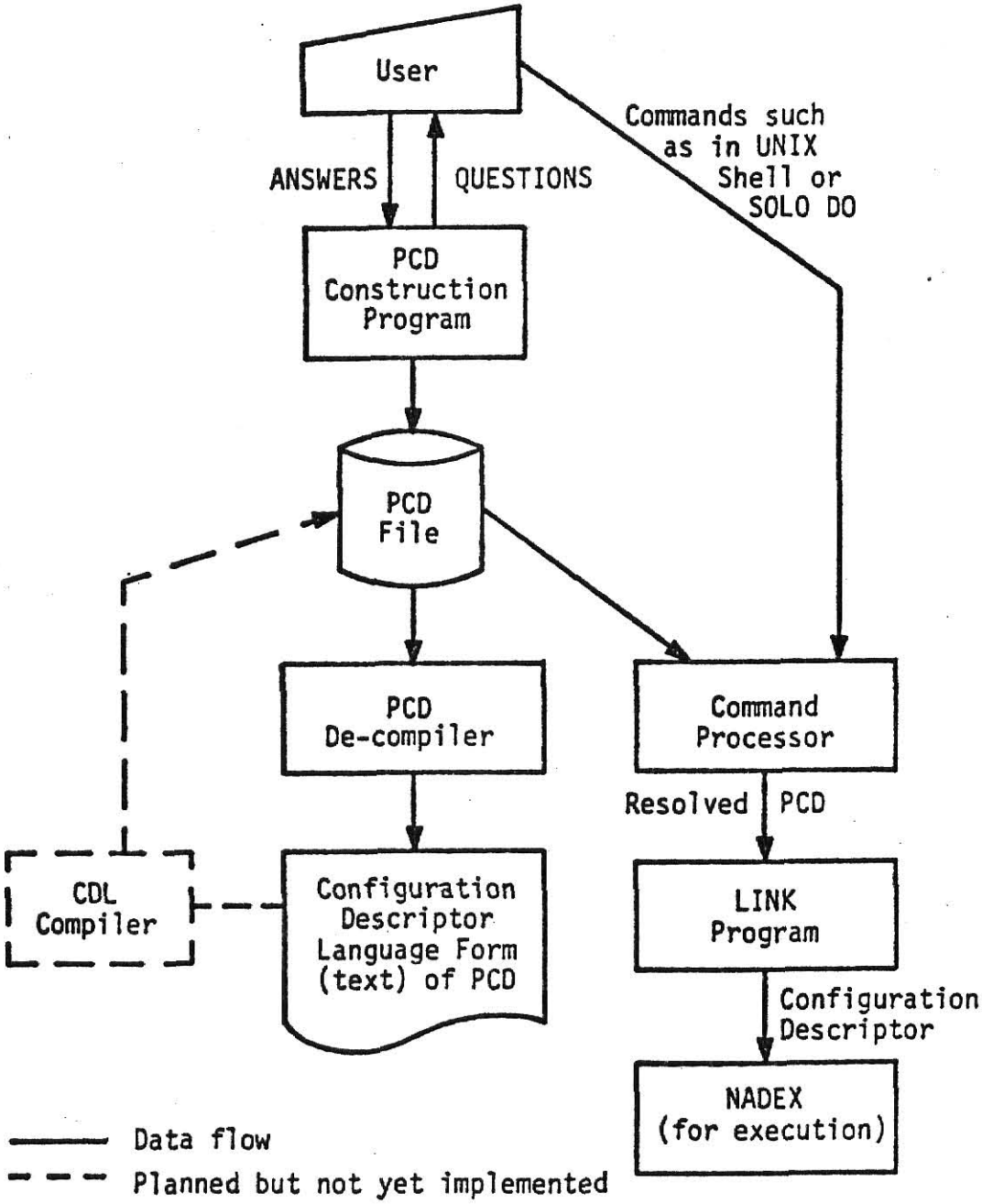


Figure 4. The PCD workbench.

the PCD LINK Program.

The Interactive PCD Constructor allows a user to define a new PCD record and enter it into the PCD library. The PCD De-compiler allows the user to examine the contents of a PCD record as formatted text. LINK maps a PCD file which has been completely resolved by a command processor [1] into a Configuration Descriptor [3] which is then submitted to the operating system for execution [3].

A final program has been planned for inclusion in S^3T. This program will be a PCD compiler which will compile the text form of a PCD, such as that produced by the PCD De-compiler, into a PCD file. This will allow a user to edit a PCD file by de-compiling it, editing the resulting text, and then re-compiling the text into a PCD file. The relationship of these programs is shown in Figure 4.

4.2.1 Partial Configuration Descriptor Format

The Partial Configuration Descriptor (PCD) is a Pascal record which describes software configurations at the user level. A software configuration may be described using one or more PCDs. An illustration of the structure of the PCD record which describes the partial configuration illustrated in Figure 2 is contained in Figure 5.

The command processor supplies parameters to each PCD based on a template for parameters contained in the PCD parameter table. If the command processor allows dynamic connection of modules, such as the UNIX command processor, then a new PCD is created to describe the encapsulation of the specified modules. The command processor then transmits the name of the PCD of the complete configuration to LINK. LINK traverses the tree of PCD records extracting leaf nodes and copying

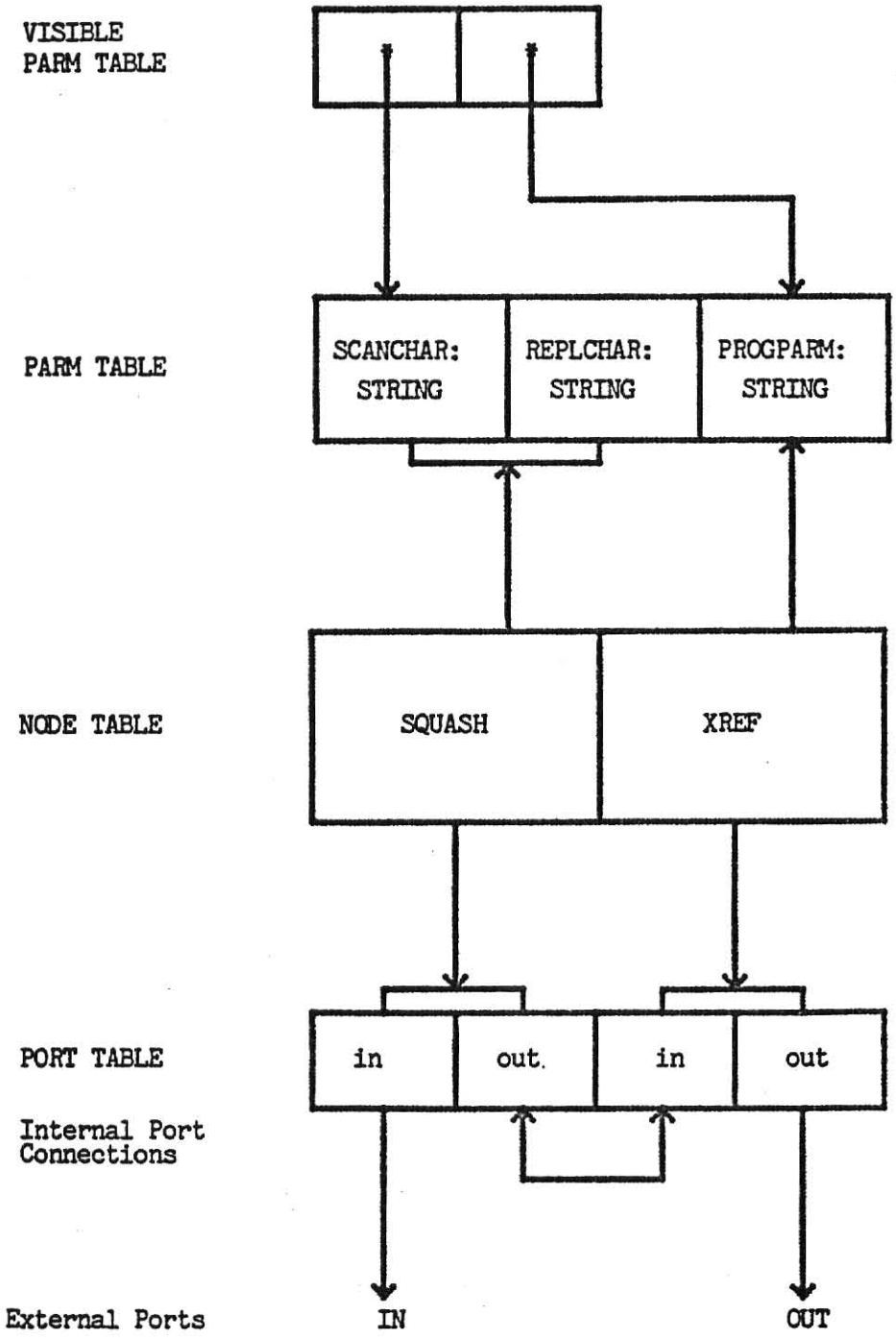


Figure 5. PCD record structure.

them into a Configuration Descriptor to be submitted to NADEX for execution.

PCD records support a user view of software configurations and provide independence and modularity of software configuration descriptions.

The identifiers in the PCD record used to name the module, nodes, and ports facilitate construction and combination of modules. A module may be abstracted by its module name, a list of its external port names, and its visible parameter names. Within a module, each port may be uniquely qualified using its node name and its port name. Connections between nodes are described using the port qualifier <nodename.portname> .

The PCD record contains three major sections: a node table which describes each node in the module, a port table containing descriptions of the ports of each node and their interconnections, and a parameter table containing types, names, and default values for three types of parameters: unresolved node fields, program parameters, and parameters to encapsulated PCDs. The node table contains an entry for each node in the module. The possible types of a node are: sequential or concurrent program, file access to use a file, IO to access a system device, subsystem to access the resources of a server subsystem such as a batch execution monitor, and module, which represents a module described by another PCD.

Each node description record contains an identifier supplied by the user which names the node. This node name must be unique within the PCD. Each entry in the node table contains a variant record which contains information for the type of that node.

User program nodes contain information for the type of prefix, Solo or Native, expected by the user program as well as the program, overlay code, and data area memory requirements.

Under the NADEX operating system, the program code executed by a node is loaded into the system process implementing the program node through a port connected to a file access node for the program code file. The program node designates which port is to be used to load the program. Typically, this port will be connected to a file access (FA) node which has access to the program code file. An index and counter designate the start and number of parameters in the parameter table which are parameters to the user program node. Parameters of a program node include program parameters as well as parameters specifying values to be used for the program memory sizes or program code filename.

An IO node contains the address of the device and its access method, CONSOLE for interactive terminals, or SEQUENTIAL INPUT or SEQUENTIAL OUTPUT for unit record devices such as cards, tape, or paper tape.

A subsystem access node contains the well-known name of the server to whom access will be gained at initialization time.

A module node, which encapsulates another PCD, contains a file descriptor which points to the file containing the encapsulated PCD. Parameter index and count fields designate the parameters which are to be passed to the encapsulated PCD.

Every node type contains an index and count into the port table. All ports for a node appear consecutively in the port table. Nodes of type FA or IO have only one port.

The port table contains an entry for each port of each node. Each

entry in the port table contains the number of the port used by the node to access this port and an identifier supplied by the user to uniquely name the port within this PCD. The port name must be unique within the node the port occurs in so that the node name and port name will uniquely identify any port in a PCD. FA and IO node ports are not named since the name of the node will determine its only port.

Each port has associated with it a protocol consisting of six fields. These fields are mode, protocol, record length, user protocol name, minimum data buffers, and minimum parameter buffers. Mode determines the direction of information flow through the port. Possible port modes are READ, WRITE, and READ-WRITE. The protocol field describes the order and type of operations performed on the port. There are four currently supported port protocol types. ASCII indicates character operations. SEQUENTIAL indicates the sequential transmission (or reception) of a single record type. RANDOM requires the transmission of a control message before each data transfer. USER protocol allows a name to be supplied by the user to indicate that the port follows a coroutine protocol for a specific server module. Record_length identifies the logical record length for SEQUENTIAL, RANDOM, or USER protocols. ASCII protocol implies a record length of one. The two count fields, minimum_data_buffers and minimum_parameter_buffers indicate resource information for the minimum number of data and parameter buffers used by this port's protocol.

Port protocol information is provided by the author of the program using the port or by the file attributes or device characteristics for FA or IO nodes ports.

Each port table entry contains an index into the node table which

identifies the owner node of the port. A port may be designated as internal or external. Internal indicates that the port is connected to another port in this module. An external port is one which is external to the module and accessible through the list of ports in the module interface. There are two variants of the port table entry which depend on its type. If the port is internal, then the record contains the index in the port table of the remote port to which this port is connected. If the port is external the port table contains an identifier which names the external port. External port names must be unique within the scope of the PCD. If the port is owned by an FA node, then an index into the parameter table indicates the parameter containing the name of the file to be accessed by the node. The file name may be represented as either an implementation-dependent file descriptor, or an implementation-independent pathname.

A node of type module is incorporated into a PCD by using the file descriptor to extract the port table entries for the external ports and the parameter table entries for the visible parameters of the encapsulated module into the port and parameter tables of the module node.

When two ports are connected, it is necessary to verify that their protocols are compatible. The checks currently implemented are:

(1) The modes must be compatible pairs; [read, write], [write, read], or [read_write, read_write]. Also see Table 2 for another possible technique.

(2) The protocol must be the same. However, ASCII protocol is compatible with SEQUENTIAL protocol, since the SEQUENTIAL records are packed and unpacked invisibly by NADEX.

(3) The record lengths must be the same. However, ASCII (record length one) is compatible with any record length SEQUENTIAL for the reason given above.

(4) The minimum number of buffers required must be the same.

(5) For user protocols, the USER_NAME must be the same.

The last major component of a PCD record is the parameter table. This table contains the information necessary to allow the user to provide certain information at a later time. The parameter table also contains all file description information. Parameters have two attributes, VISIBLE and OPTIONAL. VISIBLE signifies that the parameter appears in the external parameter list in the interface to the module described by the PCD. In general, the only parameters which are not VISIBLE are file description parameters which are not eligible for deferred evaluation, such as those for program files. OPTIONAL parameters are those which are VISIBLE, but have default values, so that they need not be resolved later.

Each entry in the parameter table contains an index into the node table which indicates which node owns this parameter and an enumeration which describes the how the parameter is used. The possible uses of a parameter are to supply the size of program, data or overlay areas, as a program parameter which is passed to the user program when it is

initialized, as a parameter to an encapsulated PCD, or as a file descriptor.

Each parameter table entry contains an index which indicates the position of the parameter in the owner node's list of parameters. If the node is a module node, then this index points to the corresponding entry in the visible parameter table for the external parameter of the encapsulated node. An identifier contains the name of the parameter. This name may be used by the command processor to allow keyword naming of command arguments. These names also allow mnemonic prompts for arguments to be given to the user.

The parameter type variants are borrowed from SOLO ARGTYPES[6]. A tag field indicates the type of the parameter and a variant contains the default values for parameters which are VISIBLE and OPTIONAL. Possible parameter types are boolean, integer, string, file descriptor, pathname, or identifier. The format of the file descriptor type is dependent on the host file system. The visible parameter table contains pointers to all parameters which appear in the module interface. The order of entries in the visible parameter table determines the order of appearance of the external parameters. The first zero entry in the visible parameter table is considered to terminate the list of visible parameters.

Since the PCD records are Pascal records, all the tables are fixed length. The number of nodes was deliberately kept small in an attempt to prevent the construction of large, unstructured configurations. Large configurations must be constructed incrementally in a modular fashion. This reduces the impact of modifying a module description or of changing the structure of a configuration.

4.2.2 The Interactive PCD Constructor

The Interactive PCD Constructor allows a user to create PCD records and add them to the library of module descriptors in an orderly and understandable fashion. Since the general subject of human engineering of the descriptions of arbitrary graph structures has not been well researched, the Interactive PCD Constructor is a first attempt at such a tool. The Interactive PCD Constructor is intended to be robust in the sense that it will not create an inconsistent PCD file regardless of input. However it is quite easy to create a meaningless PCD. User input errors cannot be corrected. This seems quite a drawback to the current version, except that the most complex PCD takes about five minutes to re-enter.

4.2.3 PCD De-compiler

The PCD De-compiler is an Spascal program which formats a PCD file into readable text. Program nodes are displayed followed by their encapsulating PCD.

4.2.4 LINK

The LINK program maps a completely resolved PCD into a Configuration Descriptor[24] and submits it for execution. LINK is the only part of the S^3T which is host-system dependent.

The command processor passes LINK the name of a PCD file which describes a completely resolved configuration which is to be executed and receives a return message indicating if the mapping was successful.

The LINK program accepts the name of the PCD file from the command

processor. LINK then processes each node, allocating Data Transmission Streams (DTS) to connect its ports and supplying values for its parameters. The nodes are processed by starting with the root PCD file and performing a postorder traversal (left to right, from bottom up) of the entire PCD tree. Configuration Descriptor nodes are allocated as the leaf nodes of the PCD tree are traversed.

Each connection between two PCD ports is realized by a pair of simplex DTSs connecting the corresponding Configuration Descriptor ports. DTS pairs are allocated and assigned at the time that both ports of a connection are known. For ports with internal connections, DTS assignments are made at the time the first port is processed. DTS assignments for ports with external connections are deferred until the connection can be made at a higher level.

Resolution of hierarchical PCD nodes is recursive. Each module node is resolved by passing its name, port table, and parameter table recursively to procedure RESOLVE_PCD. The port table and parameter table passed contain the external ports and external parameters of the encapsulated module. This recursion ends whenever a primitive node (program, IO, subsystem interface, or file access) is found. For each primitive node except for FA nodes, a node is allocated in the Configuration Descriptor with the same number of ports as the PCD node. Every port with an external connection is located in the upper level port table by matching the external name of the lower level PCD port with a port name in the upper level port table. The identity of the Configuration Descriptor port which has been allocated is propagated upward one level.

File access node ports are allocated from the top of the

Configuration Descriptor port table backwards so that the ports of FA nodes can be processed as they are found and then kept together so they can all be ports of a single subsystem interface node for the File Subsystem.

The LINK program is intended to be robust in the sense that an erroneous PCD or incorrect filename from the command processor will not cause the LINK program to terminate. It will output an appropriate error message to the console, return an error result, and wait for another request.

4.3 EVOLUTION OF S^3T

When the NADEX operating system[24] became operational in April 1979, it had no command processor or job control system. The NADEX system accepts Configuration Descriptors which describe software configurations to be executed. This descriptor consists of an array of nodes and their attributes with their interconnection represented by integers identifying each simplex connection for each of two message channels for data and parameter buffers. Originally, configurations were created by a Sequential Pascal program executing in a fixed configuration under NADEX which assigned constants into a record variable of Configuration Descriptor type and then submitting the record to NADEX for execution. This technique was almost intolerable, since it required recompilation of the driver program to modify a configuration. It also was not amenable to human users, who do not like to communicate at this low level. However unwieldy, programs which created various configurations proliferated.

The need for parameters for filenames prompted the creation of the

first command processor. The configuration descriptor building programs were modified to write their configurations into disk files. Appended to the configuration descriptor in the file was a SOLO ARGLIST[6] which contained the names and types of the parameters and a pointer to their location in the Configuration Descriptor record. The command processor accepted a command from the user, read the corresponding Configuration Descriptor file record in, and used the information in the arglist to parse the command parameters and insert them into the Configuration Descriptor record, when was then submitted to NADEX for execution.

Even this rudimentary command processor demonstrated the enhanced help information possible with S^3T. Before long, a fair-sized library of these files existed. Then the implementer of NADEX decided that the Configuration Descriptor record format needed to be changed and the files could no longer be used. It was immediately decided that an implementation-independent representation for software configurations was needed.

The basic structure of the PCD record was taken from the Configuration Descriptor. It was decided that, due to the static nature of Pascal records, it would be necessary to allow the incremental construction of configurations (since the change to the Configuration Descriptor type which prompted the development of PCDs was to increase the number of nodes). It was also decided to raise the interconnection of modules to the user level, so named ports were added. Two ports could be connected together. A connection was bidirectional and included both message types supported by NADEX. Port protocols were added to allow typing of connections during the incremental construction of configurations. The protocol attributes MODE, PROTOCOL, and

RECORD_LENGTH were taken from the File Subsystem of NADEX [24].

Also, about this time, the PCD De-compiler program was written which formats a PCD record into readable text. The module specifications given in this report were produced by the de-compiler program.

A LINK program was written to map PCD file records (which were still being created by those SPascal programs) into Configuration Descriptor records which could be used by the existing command processor. A tool for the specification of PCD files, the Interactive PCD Constructor, was then written.

The initialization of work on a UNIX command processor for the dynamic construction of configurations prompted changes to the PCD record to add external ports and allow the encapsulation of one PCD as a node in another. The LINK program was made part of the command processor configuration, so that the UNIX command processor could restrict its scope to PCD records. The command processor created a new PCD record representing a complete configuration by encapsulating those modules specified in a UNIX pipeline command. This PCD record was then given to LINK.

One additional change was made to the PCD record format in order to add external module parameters so that command parameters could be passed down more than one level. Also at this time. the file descriptor fields were modified to allow a pathname in addition to a file pointer, and host names for nodes were added in anticipation of the completion of a distributable NADEX system.

5. SUMMARY AND CONCLUSIONS

5.1 EXPERIENCES

So far, the only users of S^3T have been the implementers of the NADEX system. The best-liked feature of S^3T is the ability to produce complete interface information for the user at configuration time. This information may be in the form of HELP or may be used to prompt for information.

The incremental construction of configurations is a very natural feature. So far, there has been some difficulty with the users not understanding port protocols, but usually, even with user specified protocols, the system errs on the side of not allowing legal connections. The time lapse between writing a module and completely forgetting what it does seems to be about two weeks. After that, even the author is dependent on the accuracy of the information in the PCD record.

The use of the Interactive PCD Constructor [20] has been another area of interest. It was found necessary to provide the user with more information than was originally anticipated. The Interactive PCD Constructor has become easy and fairly natural to use.

The existing PCD library consists of 44 PCD record files, 32 of which encapsulate one of 29 programs. The remaining 11 PCD record files contain only module nodes and describe partial or complete configurations. Experience with updating the PCD records to reflect

program changes has demonstrated the value of individually encapsulating programs. If the program specification changes but the interface is not modified, then any modification is transparent to the rest of the system.

5.2 PORTABILITY OF S^3T

This section describes how the Software System Structuring Tool might be converted for use with a message-based system other than NADEX. The target system is to be the UNIX operating system.

First, assume that a standard Pascal compiler is available. The Interactive PCD Constructor can be converted to access the UNIX file system so that new PCD record files may be constructed. The PCD records already contain fields for absolute pathnames, which are considered, at least for this example, to be implementation-independent. There are several fields in the PCD used to describe attributes of program nodes required for NADEX which are not needed by the UNIX system. These include the program node types Sequential and Concurrent, although other languages may be suitable. The overlay area memory size and the prefix type are also not needed. A necessary addition is the pathname of the code file to execute. It will also be necessary to modify the Interactive PCD Constructor so that it does not automatically create a file access node for loading the program code for a program node.

The existing NADEX command processors will be usable by interfacing it to the UNIX file system. After a configuration-building command has been processed, a complete PCD which incorporates a hierarchical description of the software configuration, including the pathnames of the data files accessed and the program files executed, is

produced by the command processor.

A new program, duplicating the function of LINK, will access the PCD and create pipes and fork processes which correspond to the nodes and connections of the PCD. Each forked process is passed pointers to the program file to be executed and the pipes or files to which its ports are connected. Note that the typing facility of S^3T has been fully utilized to prevent illegal connections, even though UNIX is untyped.

5.3 DIRECTIONS FOR FURTHER RESEARCH

Many of the message-based systems support channels allowing multiple writers and one reader. This increases the flexibility of configurations since the number of modules a coroutine server may service is not bounded by the number of ports in the server module. This facility, along with the investigation of type-safe connections in such an environment, deserves further attention.

The human engineering aspects of the construction of configurations also needs further investigation. Graphical construction techniques are the most appealing.

The best way to simplify the user interaction necessary to specify the encapsulation of programs will be to allow the compiler the generate the interface description. Also, the implementation of port type protocols using path expressions will require extensive compiler modifications.

The sufficiency of path expressions to specify port types, and their ability to validate connection-time port compatibility needs further investigation. An associated problem is allowing the type-safe

reconfiguration of executing configurations.

Desirable extensions to S^3T include the addition of machine-readable text to describe the function of each module, and the creation of a module cross-referencing facility which will report the references between modules. This will simplify the identification of modules which reference modules whose interface specification has been changed.

APPENDIX 1

AN EXTENDED EXAMPLE: THE DINING PHILOSOPHERS

INTRODUCTION

The Dining Philosophers problem [12] is an example commonly used to illustrate program structuring techniques. There are five philosophers who spend their time alternately between eating and thinking. When a philosopher gets hungry, he enters the room and sits down at a circular table. In the center of the table is a large bowl of spaghetti. There are five forks on the table, one to the left of each philosopher. The philosopher picks up the fork to his left and tries to eat. The spaghetti is so entangled that he must also pick up the fork to his right in order to eat. When a philosopher is done eating, he puts down the forks and leaves the room.

The aspect of the dining philosophers problem considered here is that of describing solutions for this problem using message-based systems. First, Hoare's implementation for his Communicating Sequential Processes (CSP) language [12] is given. Then an extension of CSP allowing ports [15] is illustrated. A discussion follows about various techniques for transmitting and processing requests.

Finally, two complete implementations are described. The first is in a proposed language, Extended Concurrent Pascal. This langauge features independent compilation and port types defined using the path expressions proposed in Chapter 3. The second implementation is that

using S^3T to describe its structure and executing under the NADEX operating system using programs written in Sequential or Concurrent Pascal [6] having prefixes which extend the language with port operations.

Figure 6 contains an illustration of the complete configuration of the dining philosophers whose implementation is described. To simplify the examples, there are only three philosophers with their forks. Each philosopher sends messages to the room to request ENTRY and EXIT, and then sends messages to each fork in turn conveying PICKUP and PUTDOWN requests. Each fork can be picked up by one of two philosophers. Reaching across the table to pick up a fork is not allowed. Any of the philosophers may ENTER or EXIT the room. The room keeps a count of the number of philosophers in it, and in some of the sample implementations, denies entry to the last philosopher to avoid the possibility of each philosopher picking up one fork and then waiting forever for the other fork.

CSP SOLUTIONS

Hoare [12] presented the solution to the dining philosophers problem shown in Figure 7. This CSP program uses typed signals to convey the requests of the philosophers to the forks and room.

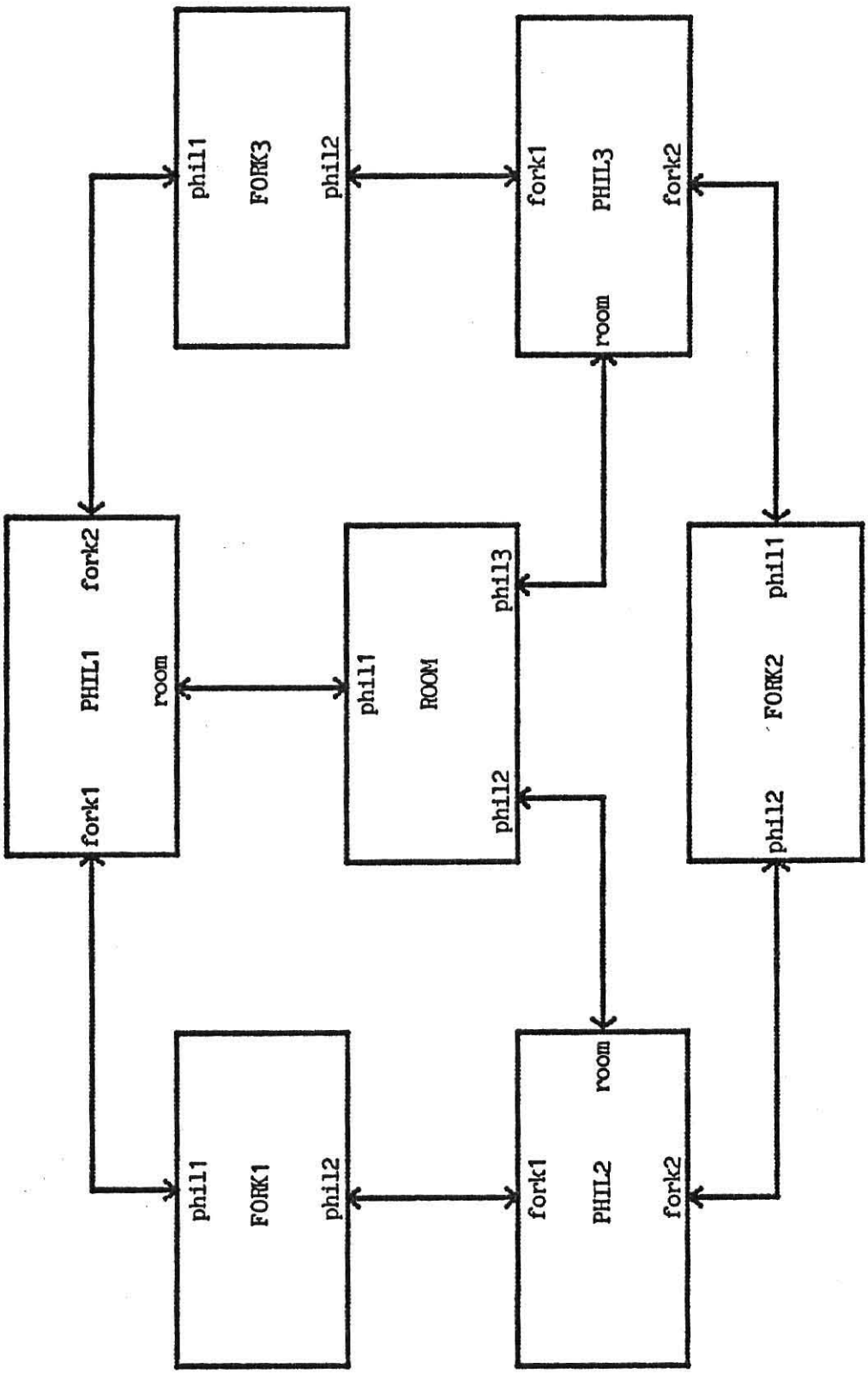


Figure 6. Dining philosophers configuration.

```
PHIL = *[ ...during ith lifetime... ->
          THINK;
          room!enter();
          fork(i)!pickup(); fork((i+1) mod 5)!pickup();
          EAT;
          fork(i)!putdown(); fork((i+1) mod 5)!putdown();
          room!exit();
          ]

FORK = *[phil(i)?pickup() -> phil(i)?putdown()
        # phil((i-1) mod 5)?pickup() -> phil((i-1) mod 5)?putdown()
         ]

ROOM = occupancy:integer; occupancy := 0;
       *[(i:0..4)phil(i)?enter() -> occupancy := occupancy + 1
       # (i:0..4)phil(i)?exit()  -> occupancy := occupancy - 1;
        ]

[room::ROOM || fork(i:0..4)::FORK || phil(i:0..4)::PHIL]
```

Figure 7. CSP solution of dining philosophers problem.

In Figure 7, square brackets ([ ]) delimit a process, the asterisk (*) denotes repetition, the number sign (#) indicates alternation, and question mark (?) and exclamation mark (!) represent output and input operations, respectively. An input or output operation names a process and a target variable within that process. The arrow indicates a guard on the command *which must* be true before the command is executed.

The problem with the CSP implementation is that CSP is not separately compilable. The addition of ports and separate compilation to CSP has been proposed [15]. Figure 8 contains an illustration of the dining philosophers problem implemented in CSP extended with ports.

```
PHIL = *[ ...during ith lifetime... ->
          THINK;
          !enter();
          !pickup1();  !pickup2();
          EAT;
          !putdown1();  !putdown2();
          !exit();
          ]

FORK = *[?pickup1() -> ?putdown1()
       # ?pickup2() -> ?putdown2()
         ]

ROOM = occupancy:integer; occupancy := 0;
       *[(i:0..4)?enter(i)() -> occupancy := occupancy + 1
       # (i:0..4)?exit(i)()  -> occupancy := occupancy - 1;
         ]

    [r::R || f(i:0..4)::F || p(i:0..4)::P;
     (i:0..4) r.enter(i) <> p(i).enter;
     (i:0..4) r.exit(i)  <> p(i).exit;
     (i:0..4) fork(i).pickup1  <> p(i).pickup1;
     (i:0..4) fork(i).putdown1 <> p(i).putdown1;
     (i:0..4) fork(i).pickup2  <> p((i+1)mod5).pickup2
     (i:0..4) fork(i).putdown2 <> p((i+1)mod5).putdown2]
```

Figure 8. CSP extended with ports.

In Figure 8, the input/output statements reference ports instead of processes. For instance, the philosopher process has six ports, ENTER, EXIT, PICKUP1, PICKUP2, PUTDOWN1, and PUTDOWN2. Since only signals are sent across ports, each request, such as pickup or putdown a fork, must have its own port. After declaring instances of the processes, the process ports are interconnected using the <> operator. Following the intentions expressed in [12], only two ports may be interconnected.

DISCUSSION OF COMMUNICATION AND SCHEDULING OF REQUESTS

In CSP, message channels are unbuffered. This means that a process executing an input operation must wait for the corresponding

process to perform an output operation. This waiting causes the processes to synchronize. In other systems, messages are buffered. If two processes wish to synchronize, it is necessary to exchange messages, i.e. a request and a response. If a process cannot be granted a specific resource, i.e. entry into the room, it must be made to wait. In CSP this means that the room cannot input the ENTER request from the process to be delayed, since the acceptance of this request implies that entry has been granted. In the buffered systems, the process will send a request for entry, and then wait for a response indicating that entry has been granted, or possibly that the request has been refused.

If a module provides services to other modules, such as the fork, it has no way of knowing which user will request a service, i.e., which of the two philosophers will request to pickup the fork first. If it does a normal input operation on one of two ports, it will ignore any requests from the other philosopher. There are several techiques used to allow any user request to be served. The order in which requests are processed may be controlled by the order of receipt of requests, explicitly by the serving module, or implicitly by the system underlying a particular implementation of the serving module.

If a process has only one input port [1] to which many ports may write, the process waits on the only port and processes the requests in the order in which they arrive. It is necessary that the users of the service identify themselves in their request message, so that an aknowledgement can be sent.

Other systems allow multiple-event waits. In this case read operations can be outstanding on several ports simultaneously. The first request received will awaken the waiting process. Another form of

allowing several outstanding read operations is the conditional read. If there is no message to be input, the read operation indicates this rather than waiting for a message to received. It is possible to wait for multiple responses by polling the ports with conditional reads. The multiplexing techiques of multiple-event waits and conditional reads allow the programmer of the module to schedule the order in which requests are processed.

Finally, some systems allow a module to contain several processes, such as Concurrent Pascal [6] or CSP [12]. In this case, one process may wait on each port for a request to arrive. The order in which requests are processed depends on the scheduling supplied by the underlying implementation.

SOLUTION IN EXTENDED CONCURRENT PASCAL

Extended Concurrent Pascal is a proposal for a language which can be used to explore the sufficiency of ports typed using path expressions. The language features separately compilable modules with ports. Each module definition consists of five sections.

The first section is optional and lists the names and sources of types to be imported from another module. A complete PROGRAM may also be imported, as shown in Figure 13. In this case, the parameters of the specified PROGRAM module describe the interface to the program.

```
PROGRAM port_types;

EXPORT TYPE
  fork_types = (pickup, putdown, fork_response);

  fork_user_port = PORT fork_types
                   PROTOCOL  WRITE(pickup);
                             READ(fork_response);
                             WRITE(putdown);
                             READ(fork_response) END;

  fork_port = PORT fork_types
              PROTOCOL  READ(pickup);
                        WRITE(fork_response);
                        READ(putdown);
                        WRITE(fork_response) END;


  room_types = (enter, exit, room_response);

  room_user_port = PORT room_types
                   PROTOCOL  WRITE(enter);
                             READ(room_response);
                             WRITE(exit);
                             READ(room_response) END;

  room port = PORT room types
              PROTOCOL  READ(enter);
                        WRITE(room_response);
                        READ(exit);
                        WRITE(room response) END;


BEGIN
END "port types".
```

Figure 9. Port type declarations in Extended Concurrent Pascal.

The next section is the PROGRAM header. If it is not desired to encapsulate this module, i.e. make it available to other modules, again as in Figure 13, the PROGRAM header may be omitted. The program header supplies the name of the module and its parameters. There are two types of parameters, initial parameters and ports. Initial parameters are constant parameters used to supply information to the module at its

initialization. Port parameters are the operands of port operations, i.e. READ and WRITE, within the module. The port parameters of a module are also constant parameters, but refer to a port in another module to which a reference is passed at initialization.

The next section of a module is the CONSTANT and TYPE declarations. A type declaration may be made visible outside the module by declaring it as an EXPORT TYPE. The definition of an EXPORT TYPE may be referenced in another module by its having an

IMPORT TYPE typename FROM modulename;

statement which references the desired type and its defining module by name. Processes, monitors, and classes may be declared as types, just as in Concurrent Pascal.

The next section consists of variable declarations.

The last section consists of a BEGIN - END block followed by a period. This last section is always required.

Figure 9 contains an illustration of a module whose only function is to define and export types. This module includes four type declarations for ports. The port type declaration syntax is explained in Chapter 3. The port type declarations form two pairs. The two ports within each pair are compatible with each other.

```
IMPORT TYPE fork_types, fork_port FROM port_types;

PROGRAM fork(phil1, phil2: fork_port);

TYPE fork_mon = MONITOR;
    VAR free: BOOLEAN;  hungry: QUEUE;
    PROCEDURE ENTRY pickup;
      BEGIN IF NOT free THEN DELAY(hungry); free:=FALSE; END;
    PROCEDURE ENTRY putdown;
      BEGIN
      free:=TRUE; IF NOT EMPTY(hungry) THEN CONTINUE(hungry);
      END;
    BEGIN free:=TRUE; END "fork_mon";

TYPE forkprocess = PROCESS(fork: fork_mon; phil: PORT);
    VAR request: fork_request;
    BEGIN
      CYCLE
      READ(phil, request);
      CASE request OF
        pickup: fork.pickup;
        putdown: fork.putdown;
      END; "case"
      WRITE(phil, request);
      END "cycle"
    END "forkprocess"


VAR fm: fork_mon;
    forkphil1, forkphil2: forkprocess;
BEGIN
  INIT fm;
       forkphil1(fm, phil1);
       forkphil2(fm, phil2);
END "fork".
```

Figure 10. Fork program in Extended Concurrent Pascal.

Figure 10 contains an illustration of the fork program. A monitor is used to allocate the fork, which may be picked up or put down. Two processes listen to their respective ports for requests to pick up the fork, and then request that operation from the fork monitor. If the fork is not available, then the requesting fork is delayed on the queue variable HUNGRY. The interface of the FORK module consists of

ports for two philosophers who may PICKUP or PUTDOWN the fork.

```
IMPORT TYPE room types, room port FROM port_types;

PROGRAM room(num_phils: INTEGER;
             phils: ARRAY[1..num_phils] OF room_port);

TYPE room_mon = MONITOR(capacity: INTEGER);
    VAR occupancy: INTEGER;  waiting  : QUEUE;
    PROCEDURE ENTRY enter;
      BEGIN  IF occupancy = capacity - 1 THEN DELAY(waiting);
             occupancy := occupancy + 1;
      END;
    PROCEDURE ENTRY exit;
      BEGIN  occupancy := occupancy - 1;
             IF NOT EMPTY(waiting) THEN CONTINUE(waiting);
      END;
    BEGIN  occupancy := 0; END "room_mon";


TYPE roomprocess = PROCESS(room: room_mon; phil: room_port);
     VAR request: room_types;
     BEGIN
       CYCLE
         READ(phil, request);
         CASE request OF
           enter: room.enter;
           exit: room.exit;
         END; "case"
        WRITE(phil, room_response);
       END "cycle";
      END "roomprocess";


VAR rm: room mon;
    roomphil: ARRAY[num_phils] OF roomprocess
BEGIN
  INIT rm(num_phils);
    FOR phil_id := 1 to num phils DO
      INIT roomphil[phil_id](rm, phils[phil_id]);
END "room".
```

Figure 11. Room program in Extended Concurrent Pascal.

Figure 11 contains a listing of the room program. It is structured similarly to the fork program, with a process listening at each port for ENTER and EXIT requests. The room monitor keeps a count

of the number of philosophers in the room, and if the third philosopher attempts to enter, delays him so that there are never more than two philosophers in the room. The interesting thing about the ROOM module is that it has a dynamic array of ports as a module parameter. NUM_PHILS is an initial parameter which describes the number of ports to configure. This number of processes are declared and initialized with access to the room monitor and one of the ports.

```
IMPORT TYPE fork_types, room types,
            fork_user_port, room_user_port FROM port_types;

PROGRAM phil(fork1, fork2: fork_user_port; room: room_user_port);
 VAR room_response: room_types; fork_response: fork_types;
BEGIN
  REPEAT
  "think"
  WRITE(room, enter);     READ(room, room_response);
  WRITE(fork1, pickup);   READ(fork1, fork_response);
  WRITE(fork2, pickup);   READ(fork2, fork_response);
  "eat"
  WRITE(fork1, putdown); READ(fork1, putdown_response);
  WRITE(fork2, putdown); READ(fork2, putdown_response);
  WRITE(room, exit);      READ(room, room_response);
  UNTIL "dead" 0=1;
END "phil".
```

Figure 12. Phil program in extended Concurrent Pascal.

Figure 12 contains an illustration of the philosopher program. It has three ports, one for each of two forks, and one for the room. The philosopher lives forever, entering and exiting the room, picking up and putting down forks, and eating and thinking.

```
IMPORT PROGRAM phil, fork, room;

CONST num_phils = 3;
      max_phil  = 2;
VAR phils: ARRAY[0..max_phil] of phil;
    forks: ARRAY[0..max_phil] of fork;
    rm   : room;

BEGIN
INIT rm(num_phils, [phil[0].rm .. phil[max_phil].rm]);
FOR i := 0 TO max_phil DO
  INIT fork[i](phil[i].fork1,
               phil[(i-1) MOD (num_phils)].fork2);
       phil[i](fork[i].phil1,
               fork[(i+1) MOD (num_phils)].phil2,
               rm.phil[i]);
END.
```

Figure 13. Complete dining philosophers in Extended Concurrent Pascal.

Figure 13 contains an illustration of the Extended Concurrent Pascal implementation of the dining philosophers problem. The PROGRAM modules for the room, fork, and philosopher are imported. An array of three processes is declared for each of the types philosopher and fork, as well as an instance of the room. All of the processes are initialized and the port connections are made by passing a port selector consisting of

processname.portname

to the corresponding connecting port. The constructor

[phil[0].rm .. phil[phil_max].rm]

constructs a constant array of ports with bounds corresponding to the subrange of indices specified.

[phil[0..phil_max].rm]

is an equivalent constructor.

SOLUTION USING S^3T AND NADEX.

This section describes the dining philosophers problem as it is implemented using S^3T and the NADEX operating system. The philosopher, fork, and room modules are written in either Sequential (philosopher) or Concurrent (fork and room) Pascal [6]. These languages support the prefix concept, which allow access to underlying resources. In NADEX these facilities include port operations.

Figure 14 contains an illustration of the approach taken to structure the dining philosophers problem in S^3T. There are three phil-fork modules, each containing a philosopher and his fork, and a room module. These modules are appropriately interconnected to create the complete configuration of dining philosophers. Figure 15 contains an illustration of the contents of the phil-fork module, and Figure 16 contains illustrations of how each program module for a philosopher, fork, or room is configured to execute under NADEX.

The structure definitions for the philosopher, fork, room, phil-fork, and complete dining philosophers follow in Figures 17 through 20. The listings of the Sequential or Concurrent Pascal programs with their prefixes appear in Figures 24 through 26.

It is important to note that the partial configuration descriptors and program listings shown represent a real, rather than conceptual implementation which may actually be executed.

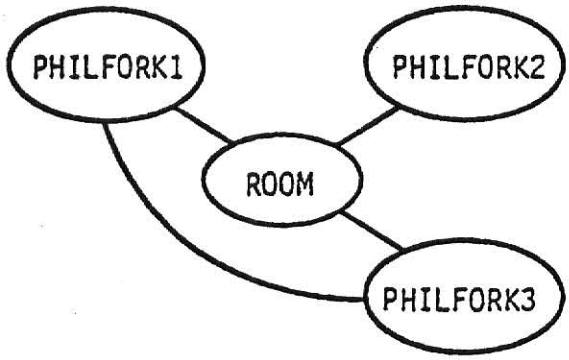


Figure 14. Dining philosophers PCD.

Figure 15. Phil-fork PCD with two external ports.

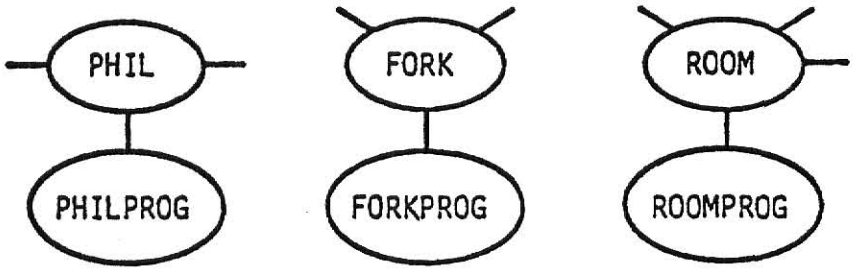


Figure 16. Program nodes and their prog loaders.

```
TYPE phil      = PROGRAM;

  PORTS
  4 progload : MODE = READ, PROTOCOL = SEQUENTIAL, RECORD_LEN = 512,
               MINIMUM DATA BUFFERS = 1, MINIMUM PARAMETER BUFFERS = 0;
  1 room     : PROTOCOL = USER, PROTOCOL_NAME = room, RECORD_LEN = 32,
               MINIMUM DATA BUFFERS = 0, MINIMUM PARAMETER BUFFERS = 1;
  2 fork1    : PROTOCOL = USER, PROTOCOL_NAME = fork, RECORD_LEN = 32,
               MINIMUM DATA BUFFERS = 0, MINIMUM PARAMETER BUFFERS = 1;
  3 fork2    : PROTOCOL = USER, PROTOCOL_NAME = fork, RECORD_LEN = 512,
               MINIMUM DATA BUFFERS = 0, MINIMUM PARAMETER BUFFERS = 1;

  ATTRIBUTES
    PREFIX = NATIVE;
    CODE_FILE = sys2:phil.img/P;
    CODE_SPACE = 1K;
    DATA_SPACE = 1K;
    OVERLAY_SPACE = 0K;

END PROGRAM;


TYPE phil      = COMPONENT;

  EXTERNAL PORTS
     room    : phil.room;
     fork1   : phil.fork1;
     fork2   : phil.fork2;

  SUBCOMPONENTS
    phil     : PROGRAM;
    philprog : FILEACCESS;

  INIT
    phil;
    philprog(FILENAME = sys2:phil.img/P, MODE = READ,
             PROTOCOL = SEQUENTIAL, RECORD_LEN = 0);

  CONNECT
    philprog. TO phil.progload;

END COMPONENT;
```

Figure 17. Partial configuration descriptor for philosopher module.

Under NADEX, each program node is implemented by a process which reads the code of the program to be executed from a port. Therefore, each module which represents a program actually consists of two nodes, one which will execute the program and the other a node which accesses the program code file. This is shown conceptually in Figure 16. The module descriptions for the philosopher, fork, and room modules in Figures 17 through 19 illustrate this explicitly.

The interfaces given for the philosopher, fork, and room modules are the same as those given in section two for the Extended Concurrent Pascal implementation.

```
TYPE fork      = PROGRAM;

  PORTS
  3 progload : MODE = READ, PROTOCOL = SEQUENTIAL, RECORD_LEN = 512,
               MINIMUM DATA BUFFERS = 1, MINIMUM PARAMETER BUFFERS = 0;
  1 phil1    : PROTOCOL = USER, PROTOCOL_NAME = fork, RECORD_LEN = 32,
               MINIMUM DATA BUFFERS = 0, MINIMUM PARAMETER BUFFERS = 1;
  2 phil2    : PROTOCOL = USER, PROTOCOL_NAME = fork, RECORD_LEN = 32,
               MINIMUM DATA BUFFERS = 0, MINIMUM PARAMETER BUFFERS = 1;

  ATTRIBUTES
    PREFIX = NATIVE;
    CODE_FILE = sys2:fork.img/P;
    CODE_SPACE = 1K;
    DATA_SPACE = 3K;
    OVERLAY_SPACE = 0K;

END PROGRAM;


TYPE fork      = COMPONENT;

  EXTERNAL PORTS
     phil1    : fork.phil1;
     phil2    : fork.phil2;

  SUBCOMPONENTS
    fork      : PROGRAM;
    forkprog  : FILEACCESS;

  INIT
    fork;
    forkprog(FILENAME = sys2:fork.img/P, MODE = READ,
             PROTOCOL = SEQUENTIAL, RECORD_LEN = 0);

  CONNECT
    forkPROG. TO fork.progload;

END COMPONENT;
```

Figure 18. Partial configuration descriptor for fork module.

```
TYPE room      = PROGRAM;

  PORTS
  4 progload : MODE = READ, PROTOCOL = SEQUENTIAL, RECORD_LEN = 512,
               MINIMUM DATA BUFFERS = 1, MINIMUM PARAMETER BUFFERS = 0;
  1 phil1    : PROTOCOL = USER, PROTOCOL_NAME = room, RECORD_LEN = 32,
               MINIMUM DATA BUFFERS = 0, MINIMUM PARAMETER BUFFERS = 1;
  2 phil2    : PROTOCOL = USER, PROTOCOL_NAME = room, RECORD_LEN = 32,
               MINIMUM DATA BUFFERS = 0, MINIMUM PARAMETER BUFFERS = 1;
  3 phil3    : PROTOCOL = USER, PROTOCOL_NAME = room, RECORD_LEN = 32,
               MINIMUM DATA BUFFERS = 0, MINIMUM PARAMETER BUFFERS = 1;

  ATTRIBUTES
    PREFIX = NATIVE;
    CODE_FILE = sys2:room.img/P;
    CODE_SPACE = 1K;
    DATA_SPACE = 4K;
    OVERLAY_SPACE = 0K;

END PROGRAM;


TYPE room      = COMPONENT;

  EXTERNAL PORTS
     phil1   : room.phil1;
     phil2   : room.phil2;
     phil3   : room.phil3;

  SUBCOMPONENTS
    room     : PROGRAM;
    roomprog : FILEACCESS;

  INIT
    room;
    roomprog(FILENAME = sys2:room.img/P, MODE = READ,
             PROTOCOL = SEQUENTIAL, RECORD_LEN = 0);

  CONNECT
    roomprog. TO room.progload;

END COMPONENT;
```

Figure 19. Partial configuration descriptor for room module.

```
TYPE PHILFORK = COMPONENT;

  EXTERNAL PORTS
     ROOM     : PHIL.ROOM;
     FORK     : PHIL.FORK2;
     PHIL     : FORK.PHIL2;

  SUBCOMPONENTS
    PHIL      : PHIL;
    FORK      : FORK;

  INIT
    PHIL;
    FORK;

  CONNECT
    PHIL.FORK1 TO FORK.PHIL1;

END COMPONENT;
```

Figure 20. Partial configuration descriptor for phil-fork module.

```
TYPE DINPHIL  = COMPONENT;

  SUBCOMPONENTS
    ROOM      : ROOM;
    PHILFORK1 : PHILFORK1;
    PHILFORK2 : PHILFORK1;
    PHILFORK3 : PHILFORK1;

  INIT
    ROOM;
    PHILFORK1;
    PHILFORK2;
    PHILFORK3;

  CONNECT
    ROOM.PHIL1 TO PHILFORK1.ROOM;
    ROOM.PHIL2 TO PHILFORK2.ROOM;
    ROOM.PHIL3 TO PHILFORK3.ROOM;
    PHILFORK1.FORK TO PHILFORK3.PHIL;
    PHILFORK1.PHIL TO PHILFORK2.FORK;
    PHILFORK2.PHIL TO PHILFORK3.FORK;

END COMPONENT;
```

Figure 21. Description of complete dining philosophers configuration.

The philosopher and fork modules are combined together by the partial configuration descriptor PHILFORK. There are three ports for this module. One refers to the other fork which the philosopher can use. The second refers to the second philosopher which can pick up his fork. The third allows access to the room.

Figure 21 describes the structure of the complete dining philosophers problem. Figure 22 contains an illustration of the complete dining philosophers configuration as it has been described. Notice the areas of encapsulation which correspond to the descriptions provided by the S^3T module descriptors.

Figure 23 contains an illustration of the structure hierarchy of the partial configuration descriptors used to describe the dining philosophers problem.

Figures 24 through 26 contain listings of the programs for the philosopher, fork, and room as they execute under the NADEX operating system. The structure of each program parallels that described for the implementation using Extended Concurrent Pascal. Each program has a prefix which defines types and operations for ports. Within the program, ports are referenced by a constant integer. This integer is mapped into a name by S^3T.

Finally, when the dining philosophers configuration, structured as shown in Figure 22, is executed under NADEX, the nodes which access files are mapped into connections to the NADEX File Subsystem. Figure 27 contains an illustration of the actual distribution and relationships of processes during the execution of the dining philosophers configuration by NADEX.

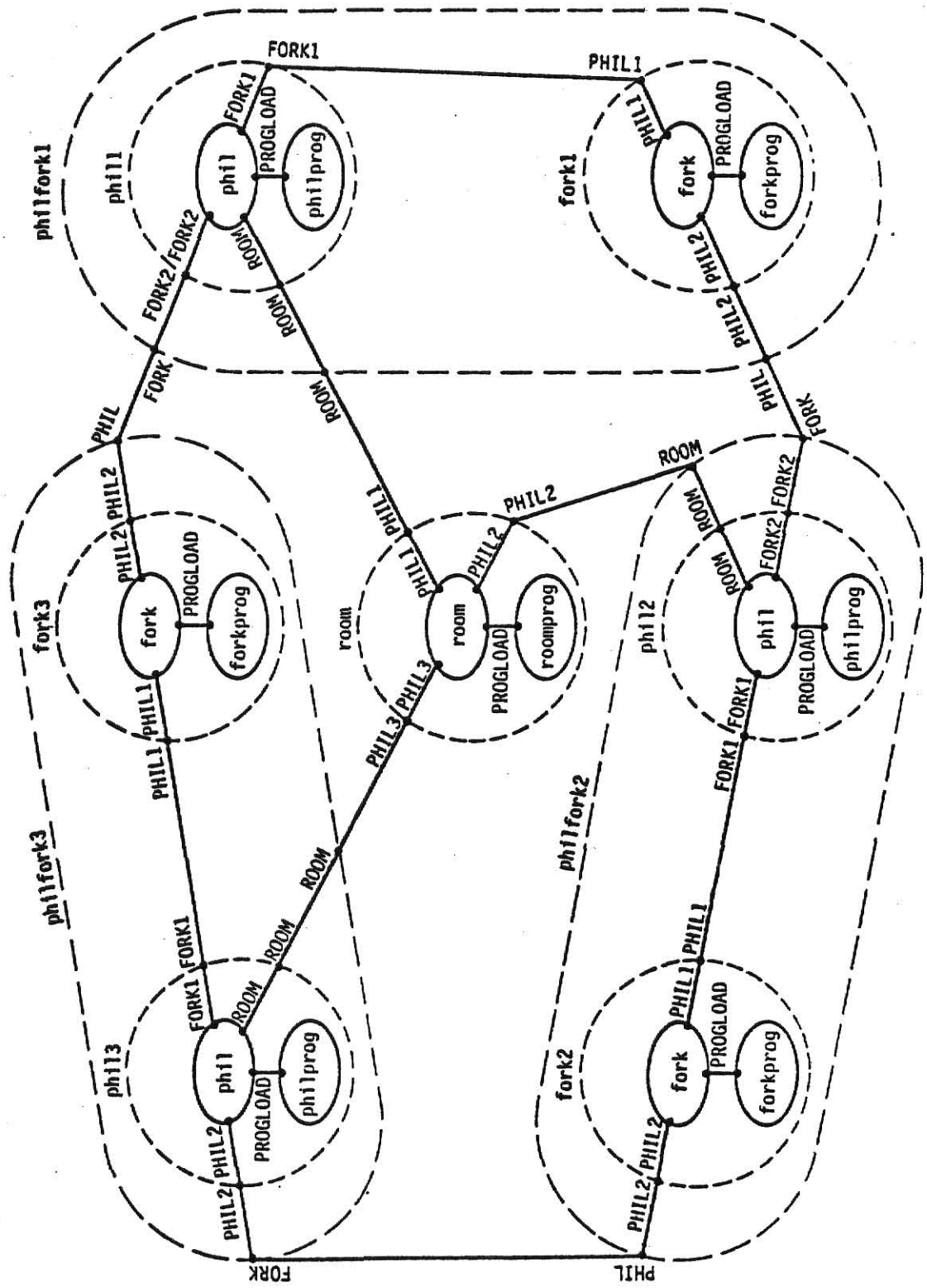


Figure 22. Complete decomposition of dining philosophers PCD.

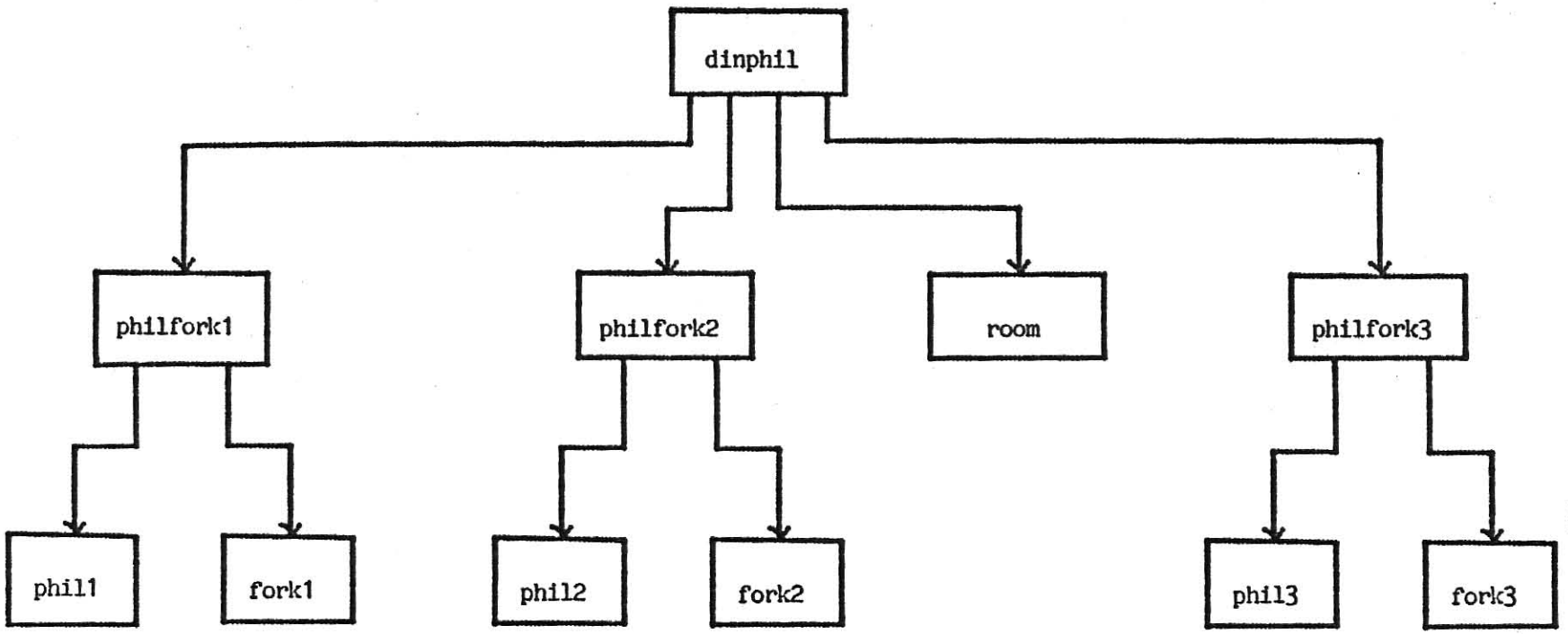


Figure 23. Dining philosophers PCD structure.

```
  CONST PARM_SIZE = 32;      MAX_PORT = 20;
  TYPE PARAMETER = ARRAY [1..PARM_SIZE] OF BYTE;
       PORT_INDX = 1..MAX_PORT;
       REQ_CODES = (REQ_OK, REQ_NODE_ABORT, REQ_DTS_ABORT, REQ_DEFER,
                    REQ_UNRES_DTS, REQ_PROT_ERROR, REQ_BAD_PORT);
  PROCEDURE READ_PARM (PORT: PORT_INDX; VAR PARM: UNIV PARAMETER;
                       VAR RESULT: REQ_CODES);
  PROCEDURE WRITE_PARM (PORT: PORT_INDX; PARM: UNIV PARAMETER;
                      CONDITIONAL: BOOLEAN; VAR RESULT: REQ_CODES);
  "end prefix"


PROGRAM phil;


  TYPE req_type = (pickup, putdown, enter, exit);
       msg_type = RECORD
                    req: req_type;
                    rest: ARRAY[1..26] OF char
                  END;
  CONST room  = 1;  fork1 = 2;  fork2 = 3;
  TYPE port_id = room..fork2;


PROCEDURE msg(port     : port_id;  request : req_type);
  VAR req_block, response : msg_type;  result   : req_codes;
  BEGIN
    req_block.req := request;
    WRITE_PARM(port,req_block,FALSE,result);
    READ_PARM (port,response, result);
  END "msg";


BEGIN
  REPEAT
  "think"
  msg(room, enter);
  msg(fork1, pickup);
  msg(fork2, pickup);
  "eat"
  msg(fork1, putdown);
  msg(fork2, putdown);
  msg(room, exit);
  UNTIL "dead" 0=1;
END.
```

Figure 24. Prefixed Sequential Pascal program for philosopher.

```
PREFIX
    TYPE PORT_INDX = 1..20;  UNIV_PARM = ARRAY [1..32] OF BYTE;
         REQ_CODES = (REQ_OK, REQ_NODE_ABORT, REQ_DTS_ABORT, REQ_DEFER,
                      REQ_UNRES_DTS, REQ_PROT_ERROR, REQ_BAD_PORT);
  PROCEDURE READ_PARM (PORT: PORT_INDX; VAR PARM: UNIV UNIV_PARM;
                       VAR RESULT: REQ_CODES);
  PROCEDURE WRITE_PARM (PORT: PORT_INDX; PARM: UNIV UNIV_PARM;
                    CONDITIONAL: BOOLEAN; VAR RESULT: REQ_CODES);
END;


CONST phil1 = 1;  phil2 = 2;
TYPE phil_ports = 1..2;
     req_type = (pickup, putdown, enter, exit);
     msg_type = RECORD  req: req_type;
                        rest: ARRAY[1..26] OF char  END;


TYPE fork_mon = MONITOR;
    VAR free: BOOLEAN;  hungry: QUEUE;
    PROCEDURE ENTRY pickup;
      BEGIN IF NOT free THEN DELAY(hungry); free:=FALSE; END;
    PROCEDURE ENTRY putdown;
      BEGIN
      free:=TRUE; IF NOT EMPTY(hungry) THEN CONTINUE(hungry);
      END;
    BEGIN free:=TRUE; END "fork_mon";


TYPE forkprocess = PROCESS(fork: fork_mon; phil: phil_ports);
    VAR req_block: msg_type;  result: req codes;
    BEGIN
      CYCLE
      READ_PARM(phil, req_block, result);
      CASE req_block.req OF
        pickup: fork.pickup;
        putdown: fork.putdown;
      END; "case"
      WRITE_PARM(phil, req block, FALSE, result);
      END "cycle"
    END "forkprocess"


VAR fm: fork_mon;  i: phil_ports;
    serv: ARRAY[phil_ports] of forkprocess;
BEGIN
  INIT fm;
  FOR i := phil1 TO phil2 DO INIT serv[i](fm, i);
END.
```

Figure 25. Prefixed Concurrent Pascal program for fork.

```
PREFIX
    TYPE PORT_INDX = 1..20;  UNIV_PARM = ARRAY [1..32] OF BYTE;
         REQ_CODES = (REQ_OK, REQ_NODE_ABORT, REQ_DTS_ABORT, REQ_DEFER,
                      REQ_UNRES_DTS, REQ_PROT_ERROR, REQ_BAD_PORT);
    PROCEDURE READ_PARM (PORT: PORT_INDX; VAR PARM: UNIV UNIV_PARM;
                         VAR RESULT: REQ_CODES);
    PROCEDURE WRITE_PARM (PORT: PORT_INDX; PARM: UNIV UNIV_PARM;
                      CONDITIONAL: BOOLEAN; VAR RESULT: REQ_CODES);
END;


CONST phil1 = 1;  phil2 = 2;  phil3 = 3;
TYPE phil_ports = phil1..phil3;
     req_type = (pickup, putdown, enter, exit);
     msg_type = RECORD  req: req_type;
                        rest: ARRAY[1..26] OF char END;


TYPE room_mon = MONITOR(capacity: INTEGER);
    VAR occupancy: INTEGER;  waiting  : QUEUE;
    PROCEDURE ENTRY enter;
      BEGIN  IF occupancy = capacity - 1 THEN DELAY(waiting);
             occupancy := occupancy + 1;
      END;
    PROCEDURE ENTRY exit;
      BEGIN  occupancy := occupancy - 1;
             IF NOT EMPTY(waiting) THEN CONTINUE(waiting);
      END;
    BEGIN  occupancy := 0; END "room_mon";


TYPE roomprocess = PROCESS(room: room_mon; phil: phil_ports);
    VAR req_block: msg_type;  result: req_codes;
    BEGIN
      CYCLE
       READ_PARM(phil, req_block, result);
       CASE req_block.req OF enter: room.enter;
                             exit: room.exit;
       END; "case"
       WRITE_PARM(phil, req_block, FALSE, result);
     END "cycle";
    END "roomprocess";


VAR rm: room mon;
    serv: ARRAY[phil_ports] of roomprocess;
    i: phil_ports;
BEGIN
  INIT rm(3);
  FOR i := phil1 TO phil3 DO INIT serv[i](rm, i);
END.
```

Figure 26. Prefixed Concurrent Pascal program for room.

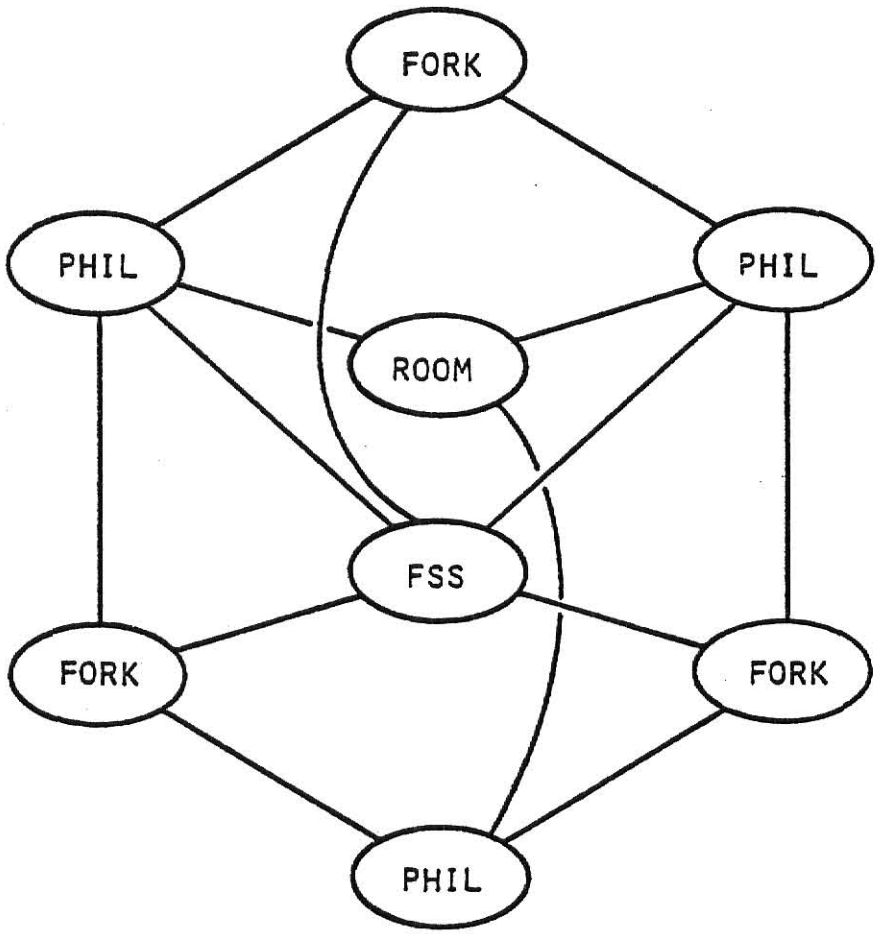


Figure 27. Dining philosophers executing under NADEX.

APPENDIX 2

SAMPLE USER SESSIONS OF MODULE CONSTRUCTION

This appendix contains examples of the way the user specifies the construction of a software configuration using S^3T. The user interacts with the Interactive PCD Construction program to answer questions about the module description being created. There are three sample sessions below. Each details the construction of one of the partial configurations illustrated in Figures 1 through 3. The first session shows the construction of the SQUASH program module. In the second session, the SQUASH program module is combined with the XREF program module to create a partial configuration. The last example shows the construction of the complete configuration by the addition of two file access nodes to the SQUASH-XREF node.

SESSION 1. CONSTRUCTION OF SQUASH PROGRAM PCD

```
NADEX PROTOTYPE INTERACTIVE CONFIGURER R00-00

     CURRENT SYSTEM LIMITS:
                              MAX NODES = 8
                              MAX PORTS = 32
                              MAX PARMS = 16
     REMEMBER THAT EACH SEQUENTIAL AND FA NODE USE ONE PARAMETER
     AND ANOTHER NODE FOR THE PROG LOADER

MODULE NAME ?
->SQUASH
NUMBER OF NODES ?
->1
FIRST NODE NAME ?
->SQ
FIRST NODE TYPE ? (SEQUENTIAL/CONCURRENT/IO/FILEACCESS/HIERARCHICAL)
->SEQUENTIAL
PROGRAM FILE NAME ?
->SQUASH
PREFIX TYPE ? (SOLO/NATIVE)
->NATIVE
PROGRAM CODE SIZE ? (IN K-BYTES)
->1
PROGRAM DATA SIZE ? (IN K-BYTES)
->1
PROGRAM OVERLAY AREA ? (IN K-BYTES)
->0
NUMBER OF PROGRAM PARAMETERS ?
->2
FIRST PROGRAM PARAMETER TYPE ? (INTEGER/BOOLEAN/STRING/ID/FILENAME)
->STRING
PARAMETER NAME ?
->SCANCHAR
PARAMETER POSITION ?
->1
OPTIONAL PARAMETER ?
->YES
DEFAULT STRING ?
->'#'
SECOND PROGRAM PARAMETER TYPE ? (INTEGER/BOOLEAN/STRING/ID/FILENAME)
->STRING
PARAMETER NAME ?
->REPLCHAR
PARAMETER POSITION ?
```

```
->2
OPTIONAL PARAMETER ?
->YES
DEFAULT STRING ?
->'^'
NUMBER OF PORTS ?
->2
FIRST PORT NAME ?
->WEST
FIRST PORT NUMBER ?
->1
FIRST PORT PROTOCOL ? (ASCII/SEQUENTIAL/RANDOM/USER)
->ASCII
FIRST PORT MODE ? (READ/WRITE/RW)
->READ
SECOND PORT NAME ?
->EAST
SECOND PORT NUMBER ?
->2
SECOND PORT PROTOCOL ? (ASCII/SEQUENTIAL/RANDOM/USER)
->ASCII
SECOND PORT MODE ? (READ/WRITE/RW)
->WRITE

ELIGIBLE PORTS:
                SQ.EAST
SQ.WEST  CONNECTS TO ? (NODENAME.PORTNAME/'EXTERNAL')
->EXTERNAL
EXTERNAL PORT NAME?
->IN
ELIGIBLE PORTS:
SQ.EAST  CONNECTS TO ? (NODENAME.PORTNAME/'EXTERNAL')
->EXTERNAL
EXTERNAL PORT NAME?
->OUT
FILE ALREADY EXISTS, TYPE 'YES' TO REPLACE OR 'NO' TO CHANGE MODULE NAME
->YES
```

SESSION 2. CONSTRUCTION OF SQX PCD

```
NADEX PROTOTYPE INTERACTIVE CONFIGURER R00-00

     CURRENT SYSTEM LIMITS:
                              MAX NODES = 8
                              MAX PORTS = 32
                              MAX PARMS = 16
     REMEMBER THAT EACH SEQUENTIAL AND FA NODE USE ONE PARAMETER
     AND ANOTHER NODE FOR THE PROG LOADER

MODULE NAME ?
->SQX
NUMBER OF NODES ?
->2
FIRST NODE NAME ?
->SQ
FIRST NODE TYPE ? (SEQUENTIAL/CONCURRENT/IO/FILEACCESS/HIERARCHICAL)
->HIERARCHICAL
HIERARCHICAL NODE NAME?
->SQUASH


        SQUASH       [SCANCHAR: STRING DEFAULT('*')],
                     [REPLCHAR: STRING DEFAULT('^')]
           EXTERNAL PORTS: IN, OUT

DISPOSITION OF PARAMETER 'SCANCHAR' ? (DEFAULT/VALUE/PARAMETER)
->VALUE
STRING VALUE ?
->'.'
DISPOSITION OF PARAMETER 'REPLCHAR' ? (DEFAULT/VALUE/PARAMETER)
->VALUE
STRING VALUE ?
->':'
SECOND NODE NAME ?
->X
SECOND NODE TYPE ? (SEQUENTIAL/CONCURRENT/IO/FILEACCESS/HIERARCHICAL)
->HIERARCHICAL
HIERARCHICAL NODE NAME?
->XREF


        XREF       [PROGPARM: STRING DEFAULT('IM,SU')]
           EXTERNAL PORTS: IN, OUT
```

```
DISPOSITION OF PARAMETER 'PROGPARM' ? (DEFAULT/VALUE/PARAMETER)
->VALUE
STRING VALUE ?
->'IM,SU'
ELIGIBLE PORTS:
                SQ.OUT
                X.IN
                X.OUT
SQ.IN  CONNECTS TO ? (NODENAME.PORTNAME/'EXTERNAL')
->EXTERNAL
EXTERNAL PORT NAME?
->IN
ELIGIBLE PORTS:
                X.IN
                X.OUT
SQ.OUT  CONNECTS TO ? (NODENAME.PORTNAME/'EXTERNAL')
->X.IN
ELIGIBLE PORTS:
X.OUT  CONNECTS TO ? (NODENAME.PORTNAME/'EXTERNAL')
->EXTERNAL
EXTERNAL PORT NAME?
->OUT
FILE ALREADY EXISTS, TYPE 'YES' TO REPLACE OR 'NO' TO CHANGE MODULE NAME
->YES
```

SESSION 3. CONSTRUCTION OF SQXREF PCD

```
NADEX PROTOTYPE INTERACTIVE CONFIGURER R00-00

     CURRENT SYSTEM LIMITS:
                          MAX NODES = 8
                          MAX PORTS = 32
                          MAX PARMS = 16
     REMEMBER THAT EACH SEQUENTIAL AND FA NODE USE ONE PARAMETER
     AND ANOTHER NODE FOR THE PROG LOADER

MODULE NAME ?
->SQXREF
NUMBER OF NODES ?
->3
FIRST NODE NAME ?
->SQXRF
FIRST NODE TYPE ? (SEQUENTIAL/CONCURRENT/IO/FILEACCESS/HIERARCHICAL)
->HIERARCHICAL
HIERARCHICAL NODE NAME?
->SQX


        SQX
           EXTERNAL PORTS: IN, OUT

SECOND NODE NAME ?
->INFILE
SECOND NODE TYPE ? (SEQUENTIAL/CONCURRENT/IO/FILEACCESS/HIERARCHICAL)
->FILEACCESS
FILE NAME ?
->PARAMETER
PARAMETER NAME ?
->INPUT
PARAMETER POSITION ?
->1
OPTIONAL PARAMETER ?
->NO
FILE MODE ? (READ/WRITE/RW/APPEND/REPLACE/UPDATE)
->READ
FILE EXCLUSION ? (SHARED/EXCLUSIVE)
->SHARED
 FILE PROTOCOL ? (ASCII/SEQUENTIAL/RANDOM)
->ASCII
THIRD NODE NAME ?
->OUTFILE
```

```
THIRD NODE TYPE ? (SEQUENTIAL/CONCURRENT/IO/FILEACCESS/HIERARCHICAL)
->FILEACCESS
FILE NAME ?
->PARAMETER
PARAMETER NAME ?
->OUTPUT
PARAMETER POSITION ?
->2
OPTIONAL PARAMETER ?
->NO
FILE MODE ? (READ/WRITE/RW/APPEND/REPLACE/UPDATE)
->WRITE
 FILE PROTOCOL ? (ASCII/SEQUENTIAL/RANDOM)
->ASCII
RECORD LENGTH ?
->512
ELIGIBLE PORTS:
                SQXRF.OUT
                INFILE.
                OUTFILE.
SQXRF.IN  CONNECTS TO ? (NODENAME.PORTNAME/'EXTERNAL')
->INFILE.
ELIGIBLE PORTS:
                OUTFILE.
SQXRF.OUT  CONNECTS TO ? (NODENAME.PORTNAME/'EXTERNAL')
->OUTFILE.
FILE ALREADY EXISTS, TYPE 'YES' TO REPLACE OR 'NO' TO CHANGE MODULE NAME
->YES
```

ANNOTATED LIST OF REFERENCES

1. Appelbe, B. and Kroening, M. Concurrent programming on microcomputers. Proc. 2nd Symp. on Small Systems, SIGSMALL 5, 2 (April 1979), 20-25.

   Extensions of UCSD Pascal to form a message-based system are presented.

2. Balzer, R.M. Ports - a method for dynamic interprogram communication and job control. Proc. AFIPS 1971 Spring Joint Comp. Conf. 38, 485-489.

   The concept of ports is introduced with the port operations CONNECT, DISCONNECT, SEND, RECEIVE and conditional RECEIVE; and the REQUEST-RESPONSE coroutine protocol.

3. Baskett, F., Howard, J.H., and Montague, J.T. Task communication in DEMOS. Proc. 6th Symp. Op. Sys. Principles (November 1977), 23-31.

   The facilities of the DEMOS operating system are described.

4. Bernstein, A. Output guards and nondeterminism in "Communicating sequential processes." ACM Trans. Programming Languages and Systems 2, 2 (April 1980), 234-238.

   Programmer controlled scheduling within a process and conditional writes are discussed. An implementation is proposed.

5. Bourne, S.R. The UNIX shell. The Bell System Technical Journal 57, 6, Part 2 (July-August 1978), 1971-1990.

   UNIX features ports, modules, and a dynamic interconnection syntax for pipelines. Connections are buffered.

6. Brinch Hansen, Per. The Architecture of Concurrent Programs. Prentice_Hall, Englewood Cliffs, N. J., 1977.

   The language Concurrent Pascal is described.

7. Campbell, R.H. and Habermann, A.N. The specification of process synchronization by path expressions. In Lecture Notes in Computer Science 16. Verlag-Springer, New York, 1974, 89-102.

Path expressions are introduced as a synchronization mechanism. A syntax and proposed implementation are presented.

8. DeRemer, F. and Kron, H.H. Programming-in-the-large versus programming-in-the-small. IEEE Trans. Soft. Eng. SE-2, 2 (June 1976), 80-86.

Presents a compilable Module Interconnection Language (MIL) intended for structuring traditional (procedures rather than processes) software systems.

9. Fundis, R.M. Command Processors for dynamic control of software configurations. M.S. Report. Dept. Computer Science, Kansas State University, Manhattan, Kansas (1980).

Describes the implementation of UNIX and MIRACLE command processors which use S^3T to dynamically structure software systems.

10. Gray, T.E. Network job control: the tower of Babel revisited. Doctoral Dissertation, UCLA (March 1979).

Proposes a syntax for dynamically connecting modules and presents the idea of a Resource Attribute Descriptor (RAD) for modules.

11. Geschke, C.M., Morris, J.H., and Satterwaite, E.H. Early experience with Mesa. Comm ACM 20, 8 (August 1977), 540-552.

Describes the binding and typing mechanisms of Mesa.

12. Hoare, C.A.R. Communicating Sequential Processes. Comm. ACM 21, 8 (August 1978), 666-677.

A single compilation language for message-based software systems featuring typed messages and guarded input commands is presented. There are no ports and connections are unbuffered.

13. Hunt, J.G. Messages in typed languages. SIGPLAN Notices, 14, 1 (January 1979).

The language LIMP is compared with other methods for structuring message-based software systems. LIMP allows multiple readers and writers per connection, and allows ports and channels to be passed between modules.

14. Jones, A. K., et. al. StarOS, a multiprocessor operating system for the support of task forces. In Proc. 7th Symp. Operating Systems Principles (December 1979), 117-127.

The facilities of StarOS are presented.

15. Kieburtz, R. and Silberschatz, A. Comments on "Communicating sequential processes." ACM Trans. Programming Languages and Systems 1, 2 (October 1979), 218-225.

The aspects of unbuffered communications channels, process termination, and interprocess communications topology in CSP are discussed. Ports with buffering and conditional writes are suggested as a partial solution.

16. Levine, J. R. and Morrison, J.P. Forum: data stream linkage and the UNIX system. IBM Systems Journal 18, 3 (1979), 470-475.

Parallels between DSLM and the UNIX system are discussed.

17. Morrison, J.P. Data stream linkage mechanism. IBM Systems Journal 17, 4 (1978), 383-408.

The facilities of the DSLM system are presented. The mechanism used to structure the system consists of assembly language macros.

18. Neal, D. and Wallentine, V. The module control system for Concurrent Pascal. Technical Report TR-78-05. Department of Computer Science, Kansas State University, Manhattan, Kansas (May 1978).

A tool for structuring and documenting Concurrent Pascal programs is presented.

19. Ritchie, D.M. and Thompson, K. The UNIX time-sharing system. The Bell System Technical Journal 57, 6, Part 2 (July-August 1978), 1905-1930.

The facilities of the UNIX operating system are described.

20. Rochat, K.L. and Wallentine, V.E. NADEX job control system implementation. Tech. Report TR-80-05. Dept. Computer Science, Kansas State University, Manhattan, Kansas (May 1980).

The implementation of S^3T is presented.

21. Sunshine, C. Interprocess communication extensions for the UNIX operating system: I. design considerations. Tech. Report R-2064/1-AF. Rand Corp, Santa Monica (June 1977).

The UNIX operating system is surveyed in relation to requirements for a network operating system. Proposed extensions are named ports and multi-event waits.

22. Walden, D.C. A system for interprocess communication in a resource sharing computer network. Comm. ACM 15, 4 (April 1972), 221-230.

A system for dynamic connection of ports is presented.

23. White, J.E. A high-level framework for network-based resource sharing. Proc. AFIPS 1976 National Computer Conf. 45, 561-570.

A model for a network protocol based on procedure calls is presented.

24. Young, R. and Wallentine, V. The NADEX core operating system services. Technical Report TR-79-11. Department of Computer Science, Kansas State University, Manhattan, Kansas (November 1979).

The services and facilities of the NADEX operating system are presented.

A SOFTWARE STRUCTURING TOOL FOR MESSAGE-BASED SYSTEMS

by

KIM LAWSON ROCHAT

B.S., Kansas State University, 1975

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1980

ABSTRACT

Interest in message-based systems which support software configurations is increasing. A software configuration is a network of processes connected together by ports, through which they communicate. The Software Systems Structuring Tool (S^3T) is an attempt to integrate the common aspects of message-based systems into a software engineering tool for the construction of software configurations.

This tool supports the typed interconnection of modules which allows the verification of the correctness and completeness of interconnection, incremental construction of configurations, and an implementation-independent structure representation. S^3T uses independently compiled modules with typed ports to construct distribution-independent configurations.

The ability to provide enhanced help information, allow the specification of parameters by position or keyword, and permit the construction of software configurations using named ports are features which the user will appreciate. In addition, this tool describes the resources needed to execute each module.

This structuring system has been implemented at Kansas State University in conjunction with the NADEX operating system.