APPENDIX E

From: H. Hunke, Software Engineering Environments, North-Holland Pub. Co., Amsterdam, The Netherlands, (1980).

RSSM/100

AN ASSESSMENT OF DREAM

William E. Riddle
Cray Laboratories Inc.
5311 Western Avenue
Boulder, Colorado 80301

The Design Realization, Evaluation And Modelling (DREAM) System is evaluated. A short history of the DREAM research project is first given in order to provide an historical context. Then, the significant characteristics of DREAM as a development environment are given, the design notation which is the basis for the DREAM system is reviewed, and the development tools envisioned as part of DREAM are discussed. In the concluding section, insights into development environments and their production which we have gained from the work on DREAM are presented and used to make suggestions for future work in the area of development environments.

INTRODUCTION

During software system development, help is critically needed in many activities, among them: 1) recording what is known and what has been decided about the system, 2) uncovering what is unknown, 3) assessing the suitability and the completeness of the (eventual) system, and 4) coordinating and monitoring the efforts of the team working on the development project.

Development environments are currently being studied as a way of providing help for these activities. A development environment is a collection of tools which provide a facilitating context in which to carry out development. The tools are typically programs, such as compilers or text editors, which augment the "powers" of the developers and ease the production of a suitable, executable version of the system. But, tools may also be *notational* and serve to augment the developers' denotational powers, or *cognitive* and serve to rationalize the development process.

In this paper, we give an assessment of the Design Realization, Evaluation And Modelling (DREAM) System as a development environment. In the next section, we give a short history of the DREAM project in order to provide some historical context. Then, in the following three sections, we discuss the significant characteristics of DREAM as a development environment, give a brief overview of the notational tool which is the basis for the DREAM system, and discuss the other tools provided (or to be provided) by DREAM. In the concluding section, we relate some of the insights into development environments and their production which we have gained from our work on DREAM.

A SHORT HISTORY

Like most systems, DREAM is a product of initial goals, prior experiences, biases

This work was performed while the author was affiliated with the University of Colorado and was supported, in part, by grant NSG 1638 from NASA Langley Research

and external "forces." In this section, we provide a brief history of the DREAM project with the intent of providing some insight into the technical and non-technical concerns which influenced the evolution of the DREAM system.

During the period 1973-1975, an automata theoretic formalism for modelling parallel systems was developed and its theoretical aspects were investigated (this work is reviewed in [Riddle 79d] and [Riddle 79e]). Under the formalism, a parallel system is considered to be a single-level collection of asynchronous components. Component interdependencies are modelled by message exchange and some components provide message buffering capabilities. The non-buffering components are modelled as sequential processes which carry out a controlled sequence of message production, transmission and reception operations. A parallel system's behavior may be described in terms of sequences of message transfers among components using a notation very similar to, but more powerful than, the notation of regular expressions. The existence of dual behavior/structure[1] notations affords the opportunity to assess the "correctness" of a particular system structure by assessing the consistency between the behavior produced by the structure and a specification of the system's desired behavior. However, the power of the formalism is such that very few consistency questions are algorithmically decidable.

Concurrent with this research, the TOPD system, developed by Peter Henderson and his colleagues at the University of Newcastle upon Tyne ([Henderson 75b]), was acquired and used as the basis for a senior-level software engineering course. The TOPD system is a development environment oriented primarily toward the implementation phase of sequential program development. It allows the development of a program to be done as a series of abstract descriptions, each of which is a finite-state model organized as a collection of data abstractions. The TOPD notation allows the description of behavior in terms of state transitions and the TOPD system provides assessment facilities [Henderson 75a] which allow the checking of the consistency between a procedure's behavioral and structural descriptions.

It was in this context that the DREAM project began, in early-1976, with the intent of preparing a prototype version of a TOPD-like system useful for the development of concurrent software systems. Sycor, Inc., provided support because the prototype was potentially useful in developing software for clusters of intelligent terminals.

The project was decidedly a research one, however, and was not driven in any way by a particular software development effort at Sycor or elsewhere. One goal was to investigate the feasibility and desirability of basing a concurrent software system development environment upon the theoretical model of parallel systems which had been developed. An equally strong goal was to provide a basis for the experimental evaluation of development environments and methodologies. Our aspirations in the latter direction followed from our feeling that experimental evaluation required the ability to perform well-defined, analyzable, partial development efforts and our belief that a development environment supporting modelling provided this ability.

Initially, our efforts focused upon developing a notation primarily founded upon our model of parallel systems but which utilized some notions from general systems theory and incorporated some compatible concepts from the TOPD Notation. We blatantly stole the general structure of the TOPD notation, as well as its concept of state-based models for describing data abstractions, but used our model of parallel systems as the conceptual basis for our notation and extended the state-based model adopted from TOPD. The result was the DREAM Design Notation (DDN) which will be discussed in a subsequent section.

[1] We use the term "structure" in the automata theoretic sense of denoting the causes of a system's behavior and we use the term "organization" to denote the more physical aspects of a [illegible] Thus, we would say "data organization" instead of "data structure."

During its development, we used DDN in several description tasks in order to assure ourselves that it was both effective and reasonably natural for describing a relatively broad variety of systems including operating systems, process-based problem solving systems and embedded control systems. Most of our description "experiments," however, concerned operating systems or parts of operating systems.

We also adopted the TOPD system's organization and prepared preliminary versions of many of the components of a prototype DREAM system — a data base, a syntax checker and a command interpreter. Integration testing of these components was precluded by the disbanding of the research group in late-1977. Several of us moved to other institutions, and we took advantage of the interruption in our activity as a group to both prepare reports and critically examine what we had done.

Over the last two years, our critical examination has been broadened to consider a number of specific topics which arose during our development of DDN. Some of these studies concerned the extension of DDN's conceptual base to a broader spectrum of systems ([Wileden 78a], [Segal 80]). We also investigated both the relationship of our work to system's theory [Riddle 79b] and the nature of a top-down design methodology based on dual behavior/structure notations such as DDN [Riddle 79c]. Finally, we have been developing algorithms that can be used (in modified form) to assess the consistency of DDN descriptions ([Bristow 79], [Stavely 79]).

At the moment, the research group is distributed and loosely-coupled. Each of us has a different focus to our individual work (flight control system design, distributed problem-solving system development, analysis algorithm development), but our work is somewhat integrated through our previous joint efforts.

THE DREAM SYSTEM

A wide variety of systems may be called development environments — even current-day operating systems are, in some sense, development environments. The different possibilities may be distinguished by characteristics such as the system's methodological base, the development phases supported, the intended audience, etc. In this section, we "define" the DREAM system in terms of a number of these distinguishing characteristics.

Concurrent Systems

As indicated previously, DREAM is oriented toward the development of concurrent systems, i.e., systems having parts which either actually or logically operate in parallel. We have not restricted attention to any specific type of concurrent systems and feel that DREAM is applicable to a wide variety including multiprogrammed systems, multiprocess systems, multiprocessor systems, networks and distributed systems. In DDN, we have provided a set of description capabilities which are appropriate for conceptualizing systems of any of these types even though these types differ extensively with respect to some characteristics. As a corollary, DDN does not have facilities for describing those aspects, such as the allocation of processes to processors, which distinguish these different types of concurrent systems (although these aspects can sometimes be indirectly described).

Our orientation toward this general class of software is not solely because of our previous work on a formalism for describing parallel systems. Rather, it is our belief, confirmed by our own and others' experiences, that the development of concurrent systems is particularly taxing, especially with respect to the assessment of suitability.

Design Phase

DREAM is oriented toward the *architectural design* phase of software development. In this early segment of the total design phase, the task is to delineate the system's modules and define the couplings among modules.' Thus, attention is primarily upon decomposing a system into its parts, defining the parts' interfaces, and indicating the interactions and interdependencies among the parts.

The architectural design phase is preceded and followed by other phases -- indicating what these entail serves to delimit the architectural design phase even further by indicating what it does *not* entail. Preceding the architectural design phase are the *requirements definition* and *specification* phases during which the system's overall capabilities are prescribed. Following the architectural design phase are the *algorithm design* and the *system implementation* phases during which the information structuring and manipulation aspects of the system are detailed and encoded in some executable form. Thus, sandwiched as it is between these other phases, a primary purpose of the architectural design phase is to transform the requirements levied against the entire system into information retention and manipulation requirements to be satisfied by the system's components.

Implicit in this view of the system lifecycle is that system certification and maintenance are *not* separate phases. Rather, certification, by either verification or validation approaches, is viewed as a continual concern which must be performed during *every* phase. Maintenance is viewed as regression to some previous point in the development followed by re-development. Thus, the activities of certification and maintenance are improtant during the architectural design phase and it is our intent to support these activities in DREAM.

Total System

In many applications, the components of a concurrent system are physical ones (e.g., aircraft engines) or human ones (e.g., a patient being operated on) as well as software. DDN allows the consideration of the total system (hardware, wetware and software) and thus permits the design of the software to be carried out with full concern given to the environment in which the software will function.

Modelling

It is our belief that the fundamental activity during architectural design is modelling — in fact, the name "architectural design" was chosen to deliberately suggest a relationship to the discipline of architecture in which the preparation of schematic, conceptual models is a paramount concern. The models developed during the architectural design of software (or buildings) are abstract representations of the actual system which omit the system's fine detail but faithfully reflect its externally observable characteristics. In these abstract representations, whatever is represented is done so to a level of accuracy and rigor that there is an adequate basis for suitability assessment. Also, the representations are in a medium in which alterations may be more easily investigated.

The analogy to architecture indicates one additional aspect of the design-level modelling of software. This is that there are at least two purposes of a model. One is that indicated above — it should reflect externally observable characteristics. The second purpose is that the model should be an adequate basis for preparing implementation plans (blueprints). DDN is intended as a medium in which models with either or both of these purposes can be prepared.

Functionality

There are many facets to a software system's suitability, roughly partitionable into functionality concerns, performance c[illegible] and economic concerns. Some work has been done on assessment with respect to performance concerns ([Sanquinetti 77], [Sanquinetti 78], [Sanquinetti 79]) at the level of the conceptual basis underlying DDN, but the orientation of DDN itself is exclusively upon the description and assessment of a system's functional characteristics.

This focus was taken in order to reduce the scope of our work to a manageable size rather than because of any feeling that performance or economic concerns are of less importance. Much to the contrary, we feel that truly effective environments must support the assessment of systems with respect to these other concerns; but we also know that facilities to allow such assessment are extremely difficult to provide.

Decision-making Support

As noted, certification is considered to be a continuous activity that is well-integrated with the activity of design preparation. It is an intent of DREAM that designers be able to not only gradually evolve a design but also be able to gradually evolve a defensible confidence in the suitability of that design. Thus, DREAM includes a number of tools which guide the decision-making process, allow the recording of decisions, and help designers determine the validity of their decisions. In DREAM, decision-making guidance is supported by tools which help designers identify unmade decisions and the information impacting the decisions. Decision recording is aided by providing appropriate notational tools (i.e., languages). Decision verification is aided by tools which allow designers to see the results, in terms of the system's functionality, of a decision *in the context* of all previous decisions.

To date, our accomplishments have fallen short of our intentions and we have fully developed only one decision-making tool, the DDN notation for recording decisions in terms of their effect on the structure and behavior of the system's components. We have developed other techniques, discussed later, for decision verification, but have not put these into a form compatible with DDN.

Methodology Independence

Because of our goal to provide a facility for the evaluation of a variety of methodologies, and because of our reluctance to posit a universally applicable methodology, we attempted to keep DREAM as free of methodological constraints as possible. This did not, however, mean to us that we could not make the system easier to use under one methodology and our proclivity toward top-down elaboration is quite apparent in the DDN language.

One effect of this decision is that DREAM does not enforce the use of any particular tools at any particular points during design evolution. Our view of the DREAM system user (which we adopted from TOPD) is that of a person who thinks off-line and then uses the system to help keep track of decisions and periodically derive information by which the logical consequences of the decisions may be deduced. As a consequence, DREAM does *not* preclude the entry of new information concerning the design which is incompatible with existing information. Rather, DREAM enforces only the rule that all information is syntactically correct and provides tools through which designers may uncover incompatibilities and inconsistences when warranted by whatever design "style" they use.

Language-based Integration

While DREAM does not achieve the integration of its tools by organizing them around a unifying methodology, it should be apparent from the discussion so far that it does follow the alternative approach to integration by basing the tools upon some common notation. The DDN language and the view of software systems which it embodies provide a common basis for defining the tools and their relationships in terms of how they construct, modify and manipulate descriptions in the language. The tools themselves make extensive use of both the syntactic and the semantic aspects of DDN.

In actuality, DDN is a collection of compatible sub-languages. Therefore, DREAM is more correctly characterized as being integrated on the basis of a family of compatible languages. Anyone who has worked with "large" programming languages will appreciate the desirability of having a collection of "small," well-defined languages where attention has been given to separation of concerns.

"Sophisticated" Users

DREAM is intended to be used by experienced design practitioners. Provision is made to represent information of interest to both requirements definers and system implementors, but it is assumed that the design practitioners will function as interpreters of the information for these other concerned parties. No attempt has been made to make DREAM usable by managers, end-system users, customers, documentors, testers, accountants, etc., all of whom have a legitimate concern in the design and its implications. DREAM can, in some instances, represent the information of interest to these agents, but the designers are again assumed to provide an interpretive interface to this information.

DREAM DESIGN NOTATION

The heart of the DREAM system is the DDN language. It embodies a formalism which facilitates the conceptualization of concurrent software systems during their architectural design. Also, it is the basis for the integration of the tools providing aid to design practitioners. Before describing these tools, it is necessary to give a brief overview of the DDN language and that is the purpose of this section. More detail on the DDN language can be found in the cited reports or inferred from the example that appears in the Appendix.

Structural Models

In DDN, a system is modelled as an hierarchical, but not necessarily tree-like, organization of components which operate asynchronously. At each level in this hierarchical decomposition of the system, components interact directly by the transmission of messages or indirectly through shared information repositories.

Components which interact by message transmission are called *subsystems* ([Riddle 77], [Riddle 78a], [Riddle 78b]). At each level, the subsystems represent the components which operate concurrently. Each subsystem's interface to its environment (i.e., the other subsystems at the same level) is defined in terms of *ports* through which messages may flow. The components within a subsystem know only about the ports — the environment of the subsystem is not visible to the subsystem's components. Likewise, the environment knows only about the subsystem's ports and cannot "see" inside the subsystem. Thus, the subsystem's interface is akin to a data abstraction interface with the ports serving a role analogous to a data abstraction's procedures.

Some of a subsystem's components are primitive, that is, not decomposible. These *control processes* govern the flow of messages through the ports and serve to distribute incoming messages to component subsystems and to collect messages from component subsystems for transmission out through the ports. Control processes may communicate directly by message transmission. Also, they may communicate via shared information repositories.

In Figure 1, we give a graphical representation of a DDN model using circles to denote subsystems, triangles to denote control processes and squares to denote

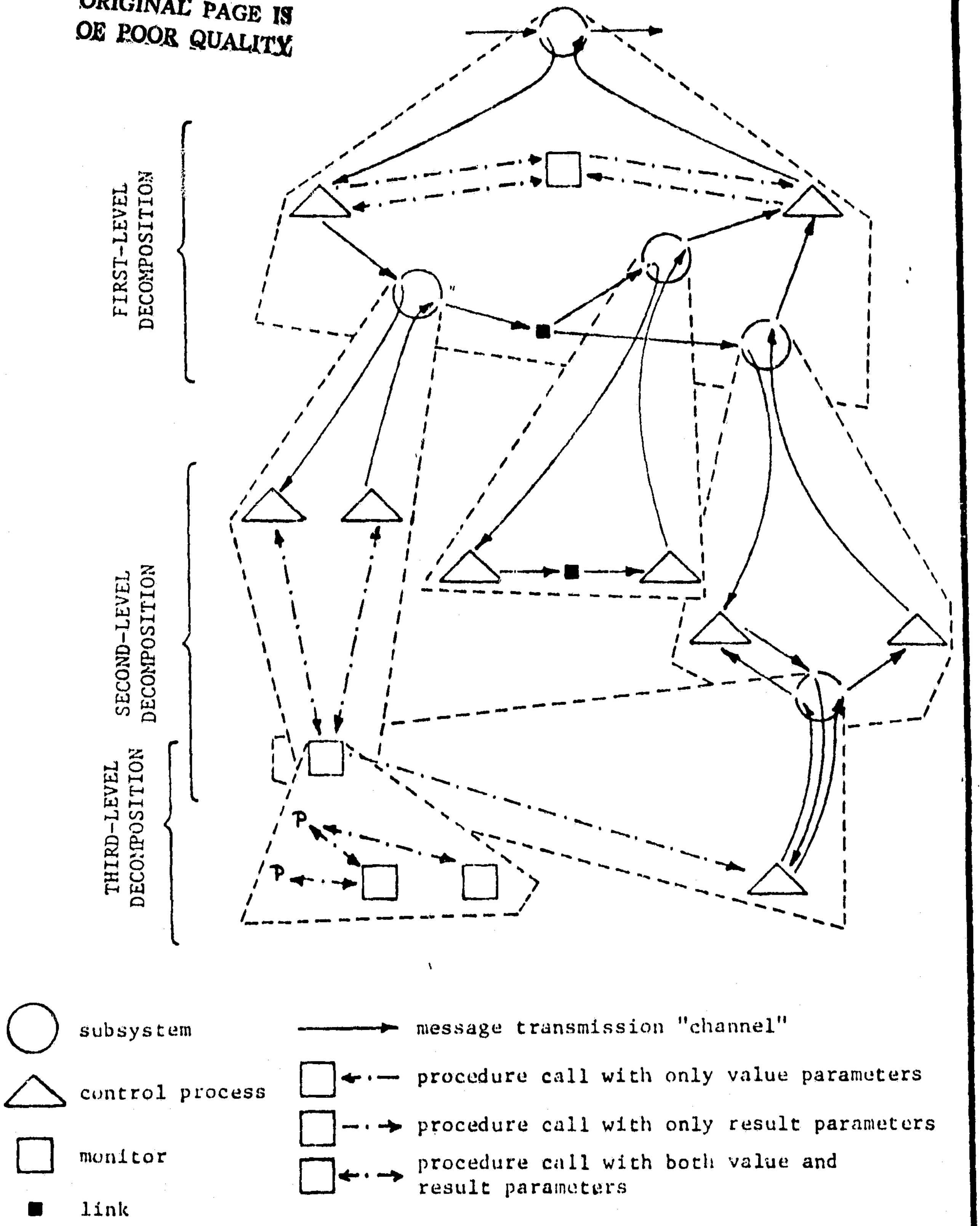
FIRST-LEVEL DECOMPOSITION
SECOND-LEVEL DECOMPOSITION
THIRD-LEVEL DECOMPOSITION
P
P
subsystem
control process
monitor
link
procedure
message transmission "channel"
procedure call with only value parameters
procedure call with only result parameters
procedure call with both value and result parameters

FIGURE 1

"skin." The small, black squares in the figure are special shared data repositories, called *links*, which provide potentially infinite message buffering facilities. This figure more clearly indicates that subsystems provide the means for encapsulating collections of data storage components (internal, shared information repositories) and data processing components (internal subsystems). Note that component-sharing is allowed, even between levels of decomposition.

Shared data repositories are modelled as *monitors* [Hoare 74] since this well-known construct provides much of what is needed [Riddle 79a]. A monitor in DDN denotes a multi-procedure data abstraction which can function as a data repository shared among asynchronous data processing components. There is the restriction that only monitors may be components within monitors. (This somewhat arbitrary restriction comes from a feeling that once a locus of control is sequentialized it should remain sequentialized.)

Behavior Models

A structural model reflects operational aspects of the system specifying interfaces and modelling the algorithms which control the use of the interfaces. This is analogous to defining the structure of a stack data abstraction and the structure of a tree-search algorithm which uses a stack by specifying the push and pop procedures and giving an abstraction of the tree-search algorithm which indicates, non-deterministically, the calls upon the push and pop procedures.

A structure gives rise to, or causes, some behavior which is the effect, over time, of "executing" the structure. Thus the behavior of a stack data abstraction is reflected by statements such as "the number of pop's is less than or equal to the number of push's," and the behavior of the stack as used by the tree-search algorithm is reflected by statements which indicate the sequences of pop's and push's which the algorithm creates when it is executed.

Notice that a component will have an *intrinsic* (or *actual*) *behavior* which is the behavior the component's structure is capable of producing. A component will also have one or more *extrinsic* (or *desired*) *behaviors* which are those stemming from the use of the component. In the absence of any knowledge of about how a component is actually used, its *required behavior* can be defined with the implication that any legal extrinsic behavior should exercise the component only in ways that are indicated in the required behavior. DREAM's certification tools, which will be discussed later, determine suitability by comparing extrinsic, required and/or intrinsic behaviors.

DDN provides a number of ways of specifying a component's required behavior and thus giving a behavioral model of the component ([Riddle 78b], [Wileden 78b]). Required behavior that relates to the usage of interfaces may be specified by indicating conditions upon the information which may legally pass through the interface. Thus, conditions may be levied against messages that may pass through subsystem ports or values that may "pass through" parameters to monitor procedure calls.

More complicated aspects of a monitor component's behavior, concerning when the component's procedures may be legally invoked, may be specified by giving pre- and post-conditions, stated in terms of the "state" of the component and associated together to form a transition which indicates the effect of the procedure. For example, a stack's states could be specified as empty, full, and otherwise and the transitions for pop could be:

full → otherwise

otherwise → otherwise or empty

which indicate that [illegible] cannot be legally invoked when the stack is empty and specify the required behavior to be "caused" when pop is invoked. (The specification of behavior by state transitions is a concept which we borrowed from TOPD.)

A final set of DDN constructs for specifying required behavior allows the description of behavior over time. *Events* can be defined as the occurrence of "interesting happenings" during system operation and the required behavior may then be expressed as required sequences of events. For example, procedure names are automatically events and the usual required behavior for a stack could be specified as:

REENTRANT(SEQUENCE(push, pop))

which, because of the semantics of REENTRANT, means that the number of push's must be greater than or equal to the number of pop's.

The DDN constructs for event sequence definition are powerful but parsimonious. Also, they are a formal notation and are therefore not extremely "friendly" for design practitioners who lack training in the formal aspects of computer science. However, they provide a set of denotational capabilities which are extremely important for behavioral modelling.

Organizational Model

Following the precedent set by the TOPD language (and originated, we believe, in the Simula language [Dahl 66]), DDN descriptions are a collection of *class* definitions. A model is therefore obtained from a description by a process of *instantiation*. Standard declarative constructs are used to denote hierarchical organization and special constructs are provided to denote component sharing since the standard declarative constructs would permit only tree-like hierarchies [Riddle 80]. Message communication pathways among subsystems are defined by using additional descriptive constructs.

DDN views instantiation as occurring entirely before execution and thus models have a static organization. Many dynamic organizations can, however, be "simulated" by the dynamic use of a static communication pathway organization, but the resulting models tend to be overly complex and unclear.

A Final Word

DDN is a poor programming language; but that's because it is not intended to be a programming language. The intent in developing DDN is to provide modelling constructs which allow the description of what modules comprise a system and what the interactions among the modules should be. Exactly *how* the modules interact is not considered to be properly part of a DDN model. Thus, for example, a single concurrent process synchronization mechanism (message transfer) is provided in DDN whereas the actual synchronization in the fully developed system would be achieved by the use of one of a number of synchronization mechanisms.

However, DDN looks like a programming language and this is perhaps a mistake. We have seen designers misled by the similarity and, as a consequence, they misuse the language. We feel, however, that the concepts included in DDN are the right ones for the architectural design of concurrent systems. We feel that the problems which arise should be solved by education of the designers rather than changing DDN.

DREAM SYSTEM TOOLS

We plan a number of tools to help design practitioners in constructing suitable models for concurrent software systems. We know that some of these tools are feasible because we have constructed prototype versions. Others have been developed as far as the algorithmic techniques are concerned but we have not yet put them in a form suitable for use with DDN. We discuss these tools in this section as if they have been constructed and the cited references indicate more exactly the degree to which they have been developed.

Data Base Core

The fundamental tool in DREAM is a data base in which are stored fragments of a DDN textual description. Most fragments define some aspect of some class of components in the model; others give information concerning tests or analyses, documentation, etc. The DDN syntax defines how these fragments are related and provides for naming them. Fragments to be retrieved from the data base are selected by these names and thus DDN itself is used as the basis for the data base query language.

Description fragments have other attributes besides names. In the current data base implementation [Humbrecht 80], there can be an arbitrary number of additional attributes although the number of attributes and their values are fixed for any data base. Further, there must be at least one attribute which hold a time/date stamp reflecting when the fragment was inserted into the data base. Users may establish a "slice" through the data base by specifying values for the attributes and any modifications to the data base are relative to the fragments in the active slice. The time/date stamp attribute is used to resolve all ambiquities, in effect selecting the "newest" fragment in any collection which is selected by an ambiguous retrieval command or slice definition. This data base organization provides a good deal of flexibility, allowing "windows" into the data base which can reflect time, versions, design team organization, etc.

Bookkeeping Tools

In keeping with the view that a DREAM user intersperses relatively long periods of offline thinking with relatively short periods of modification of the information contained in the data base, the DREAM tools for aiding the maintenance of the evolving design's description are simple extensions of the data base interface. It is assumed that the host operating system's editor can be used to prepare new or modified description fragments. The bookkeeping tools then amount to 1) an entry tool which syntax checks the fragment and presents syntactically correct fragments to the data base for insertion, and 2) a retrieval tool which interprets the user's retrieval directives and constructs the appropriate data base query commands.

Decision-making Tools

The major tools provided by DREAM are those which aid design practitioners in assessing the suitability of the design as it evolves and in identifying what remains to be designed. These tools fall into three categories which we discuss below.

Pharaphrasing Tools. Tools in this category re-present the information in the data base in a form, perhaps structured in some canonical format or presented graphically, in which the user may more easily inspect it and perhaps even identify some errors. The information is no more and no less than that already contained in the data base although the user may focus on some subset of the total information. Figure 1 is an example of what might be produced by an instantiation graph paraphrasing tool. Other paraphrasing tools could produce control maps akin to flow charts or they could produce cross-reference charts.

Extraction Tools. This type of tool examines the information in the data base and, knowing the semantics of DDN, derives feedback for the designers concerning the characteristics of the model. Usually, this feedback concerns the intrinsic simulators, finite-state testers [Henderson 75a], and event sequence expression derivers [Riddle 79e].

Consistency Checking Tools. These tools uncover incompatibilities among various description fragments or between a description fragment and some rule (e.g., no deadlock). Because of the difficulty and impossibility of doing exact, algorithmic analysis, these tools uncover anomalies (i.e., deviations which are suspicious but are noc confirmed errors) and the designers must use intuition, experience and insight to determine whether or not a detected anomaly is in fact an error. Examples of consistency checking tools are: the TOPD consistency checker [Henderson 75a], event trace checkers [Stavely 79], and synchronization anomaly detectors [Bristow 79].

CONCLUSIONS

ORIGINAL PAGE IS
OF POOR QUALITY

Even though we have not yet prepared a full implementation of DREAM, our development of DDN, along with its assessment and with our investigation into analysis techniques and other associated topics, has given us some insights into the characteristics of effective development support systems. Further, we have also gained some insight into major problems which remain to be solved on the way to achieving these systems. These insights, several of them quite obvious with 20/20 hindsight, are discussed in this concluding section.

Some Lessons Learned

Language Decomposition. By building on TOPD as a base language, and by not giving enougn thought to the inadvisability of having a large, complex language, we suffered a severe case of the "Second System Syndrome." We do not believe that we have unnecessary constructs. Nor do we feel that the constructs appear in conflicting forms. We do think that DDN should be more clearly decomposed into the family of interrelated languages which it actually is. Perhaps it is sufficient to partition the language along structure/behavior/organization lines, but that is not entirely clear at the moment.

Separation of Concerns. Decomposition of the notational tools into an integrated collection of logically complete languages is part of a larger issue. *All* tools in a development system must serve a narrowly defined purpose and it is a mistake to make tools do "double duty." For example, having DDN be both a model description language *and* the basis for a data base query language is a mistake since it complicates the language and negatively impacts the ability to easily change either aspect of the language.

It is, of course, extremely desirable that a Tower of Babel situation be avoided. The point is that our experiences with DDN indicate one should define a number of small, interrelated, well-defined languages and then attempt to amalgamate them. We made the mistake of trying to put everything into one language from the very beginning.

Tool Usage. A beautifully handcrafted banjo can be just a piece of art and not a musical instrument. So it is with software development tools -- their value depends on their usefulness for software development rather than their elegance on more esoteric levels. It is imperative, therefore, that tools emerge through a natural selection process involving actual use.

A well-established approach is to implement the tools, put them into use, and see how they are accepted. This is also an effective way to "sell" tools to practitioners since, if they accept the tool then they will ask for more (a phenomenon which has been called the "Potato Chip Principle" [Nassi 80]).

For a number of reasons, we did not take this approach in developing DREAM. Perevolve it as a result of that usage. But it is equally important to do gedanken experiments away from the heat of battle — one has only to compare Pascal and Fortran to see this. In developing DDN we have done a number of gedanken experiments and feel that this introspective, controlled usage of tools is an important development approach.

Exploration. DREAM provides little *direct* help for exploring and comparing alternative designs. Designers may describe alternatives, assess them individually and compare the results. But, more sophisticated facilities are generally needed to help developers keep track of the alternatives, their similarites and their differences. Facilities are also necessary to help developers trace back through a sequence of decisions.

Education. In teaching TOPD and DDN, we have found it possible, but difficult, to establish the right frame of mind for effective use of modelling-based tools. Even those more sophisticated students, who embrace good programming methodologies, tend to use the familiar to understand the unfamiliar and look at architectural design as something of the same nature as algorithm design, only more complicated. Further, the fact that TOPD and DDN have their heritage in programming languages fosters this reaction. We have found it mandatory to carefully establish the nature of architectural design prior to introducing design notations.

Some Problems for the Future

Education. It probably overloads the world to create yet another paraphrase of MacArthur's famous saying, but it is true that "old programmers never die they just become designers." And so it should be, since system designers must be relatively sophisticated programmers so that they create feasible designs. But this means that we cannot rely on experience being the "teacher" and must directly address the issue of educating developers in the ways to appropriately and effectively use tools and development environments.

Languages. Several proposals for primitives which should be in languages for programming distributed systems (e.g., [Andrews 77], [Hoare 78], [Liskov 79]) have duplicated some of the primitives found in DDN. This indicates that some of the primitives we developed for DDN are useful for the implementation-level description of systems. We view this as desirable since it lessens the gap between a design and an implementation. But we also feel that blurring the distinction between design and implementation is not a good idea since it will allow, and perhaps even encourage, implementation decisions to be prematurely made. If the language-extension approach to preparing development environments is chosen, we feel it is critically important to allow the use of the primitive concepts (e.g., message transmission) without having to express all the details (e.g., message buffering constraints).

Evaluation. We currently lack the techniques that will be necessary to assess the impact of tools and environments; there are many *perceived* advantages and disadvantages, but assessments are intuitive, subjective, ambiguous and contradictory. We need models of the development process and developers themselves. We need metrics of system quality and development methodology quality. And, we need some idea of how to conduct "small," experimental development projects in a way which allows valid inferences to be drawn concerning "large" efforts.

Development Style. A particular need, in order to be able to prepare truly effective environments, is for some understanding of how developers *really* carry out their work. Most development methodologies make the assumption that the process of development is, or can be, the orderly progression of logical steps, and rationalization of the process is, no doubt, beneficial. But creative processes are nortorious for their randomness and we need to understand the nature of this randomness rather than try to eliminate or control it. Only with such an understanding can effective help b [illegible] through software development tools and environments.

ACKNOWLEDGMENTS

I am indebted to the other members of the DREAM project — John Sayler, Al Segal, Al Stavely and Jack Wileden — for their help. The comments and assessments presented here were formulated during numerous discussions and I would like to thank the following people in particular for their willingness to listen and constructively criticize: Roy Campbell, Bryan Edwards, Gerry Estrin, Vic Lesser, Ed Senn, Raymond Yeh, and Pamela Zave.

REFERENCES

Andrews 77
G.R. Andrews and J.R. McGraw. Language features for process interaction. *Software Engineering Notes, 2*, 2 (March 1977), 114-127.

Bristow 79
G. Bristow, C. Drey, B. Edwards and W. Riddle. Anomaly detection in concurrent programs. *Proc. Fourth International Conf. on Software Engineering*, Munich, Germany, (September 1979), pp. 265-273.

Dahl 66
O. Dahl and K. Nygaard. SIMULA - An Algol-based simulation language. *Comm. A.C.M., 9*, 9 (September 1966), 671-678.

Dijkstra 68
E.W. Dijkstra. The structure of the T.H.E. multiprogramming system. *Comm. A.C.M., 11*, 5 (May 1968), 341-346.

Henderson 75a
P. Henderson. Finite state modelling in program development. *SIGPLAN Notices, 10*, 6 (June 1975), 221-227.

Henderson 75b
P. Henderson, R.A. Snowdon, J.D. Gorrie and I.I. King. The TOPD System. Tech. Report 77, Computing Laboratory, Univ. of Newcastle upon Tyne, England, (September 1975).

Hoare 74
C.A.R. Hoare. Monitors: An operating system structuring concept. *Comm. A.C.M., 17*, 10 (October 1974), 549-557.

Hoare 78
C.A.R. Hoare. Communicating sequential processes. *Comm. A.C.M., 21*, 8 (August 1978), 666-667.

Humbrecht 80
J. Humbrecht. DEMDAB: A design and maintenance data base system. M.S. Thesis, Dept. of Computer Sci., Univ. of Colorado at Boulder, Colorado, (to appear 1980).

Liskov 79
B.H. Liskov. Primitives for distributed computing. Proc. Symp. on Operating System Principles, Asilomar, California, (December 1979), pp. 33-42.

Nassi 80
I. Nassi. Software development tools: An industrial perspective. In Riddle and Fairley (eds.), *Software Development Tools*, Springer Verlag, Heidleberg, Germany,(1980).

Riddle 72
W.E. Riddle. Hierarchical modelling of operating system structure and behavior *Proc. A.C.M. National Conf.*, Boston, August 1972.

Riddle 77
W. Riddle. Abstract process types. RSSM/42, CU-CS-121-77, Dept. of Computer Sci., Univ. of Colorado at Boulder, (December 1977; revised July 1978).

Riddle 78a
W. Riddle, J. Sayler, A. Segal, A. Stavely and J. Wileden. A description scheme to aid the design of collections of concurrent processes. *Proc. 1978 National Computer Conf.*, Anaheim, California, (June 1978), pp. 549-554.

Riddle 78b
W. Riddle, J. Wileden, J. Sayler, A. Segal and A. Stavely. Behavior modelling during software design. *IEEE Trans. on Software Engineering, SE-4*, 4 (July 1978), 283-292.

Riddle 79a
W. Riddle, J. Sayler, A. Segal, A. Stavely and J. Wileden. Abstract monitor types. *Proc. Specification of Reliable Software Conf.*, Boston, Massachusetts, (April 1979), pp. 37-43.

Riddle 79b
W. Riddle and J. Sayler. Modelling and simulation in the design of computer software systems. In Zeigler (ed.), *Methodology in Systems Modelling and Simulation*, North-Holland, Amsterdam, The Netherlands, (1979), pp. 359-386.

Riddle 79c
W. Riddle. An event-based design methodology supported by DREAM. In Schneider (ed.), *Formal Models and Practical Tools for Information Systems Design*, North-Holland, Amsterdam, The Netherlands, (1979), pp. 93-108.

Riddle 79d
W. Riddle. An approach to software system behavior description. *Computer Languages, 4*, (1979), 29-47.

Riddle 79e
W. Riddle. An approach to software system modelling and analysis. *Computer Languages, 4*, (1979), 49-66.

Riddle 80
W. Riddle, J. Sayler, A. Segal, A. Stavely and J. Wileden. Hierarchical description of software system organization. *Proc. 13th Hawaii International Conf. on System Sci.*, Honolulu, Hawaii, (January 1980).

Sanguinetti 77
J.W. Sanguinetti. Performance prediction in an operating system design methodology. RSSM/32 (Ph.D. Thesis), Dept. of Computer and Comm. Sci., Univ. of Michigan, Ann Arbor, Michigan, (May 1977).

Sanguinetti 78
J.W. Sanguinetti. A formal technique for analyzing the performance of complex systems. *Proc. CPEUG Conf.*, Boston, Massachusetts, (October 1978).

Sanguinetti 79
J.W. Sanguinetti. A technique for integrating simulation and system design. *Proc. Conf. on Simulation, Measurement and Modelling of Computer Systems*, Boulder, Colorado, (August 1979), pp. 163-172.

Segal 80
A. Segal. Modelling supervisory systems which execute on interruptible processors. Ph.D. Thesis, Dept. of Computer and Comm. Sci., Univ. of Michigan, Ann Arbor, Michigan, (to appear 1980).

Stavely 79
A. Stavely. The membership problem for behaviors of concurrent software systems. RSSM/93 and CSR 153, Computer Sci. Dept., New Mexico Inst. of Mining and Tech., Socorro, New Mexico, (September 1979).

Wileden 78a
J.C. Wileden. Modelling parallel systems with dynamic structure. RSSM/71 (Ph.D. Thesis), Dept. of Computer and Comm. Sci., Univ. of Michigan, Ann Arbor, Michigan, (January 1978).

Wileden 78b
Wileden, J.C. Behavior specification in a software design system. RSSM/43, COINS Tech. Rep. 78-14, Dept. of Computer and Info. Sci., Univ. of Massachusetts, Amherst, July 1978.

APPENDIX:

Example DDN Description

In this appendix, we give a DDN description of an operating system that is structured similarly to the T.H.E. operating system developed by Dijkstra and his colleagues [Dijkstra 68]. The description given here is essentially a translation of the description given in a notation that served as a basis for the DDN notation [Riddle 72].

First we use the DDN Notation to give simple models of the devices managed by the operating system.

```
[memory_control_unit]: SUBSYSTEM CLASS;
   DOCUMENTATION; Items in this class of subsystems process
      requests for reading and writing of the real memory.
      Each request is in the form of an address within the
      real memory space.  No account is taken in this model
      of the response to a read request.
      END DOCUMENTATION;
   channel: IN PORT;
      BUFFER SUBCOMPONENTS; address OF [real_address]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   process: CONTROL PROCESS;
      MODEL; ITERATE; RECEIVE channel;
                END ITERATE;
         END MODEL;
      END CONTROL PROCESS;
   END SUBSYSTEM CLASS;

[operator_console]: SUBSYSTEM CLASS;
   DOCUMENTATION; This models the console and the operator
      using it.
      END DOCUMENTATION;
   message_in: INPORT;
      BUFFER SUBCOMPONENTS; message OF [program_message]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   reply_out: OUT PORT;
      BUFFER SUBCOMPONENTS; reply OF [operator_reply]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   operator: CONTROL PROCESS;
      MODEL; ITERATE; RECEIVE message in;
                      SET reply TO ans;
                      SEND reply_out;
                END ITERATE;
         END MODEL;
      END CONTROL PROCESS;
   END SUBSYSTEM CLASS;

[card_reader]: SUBSYSTEM CLASS;
   DOCUMENTATION; This models the actual card reader.  For
      purposes of this model, the card reader reads in one
      card at a time in response to requests from some part
      of the operating system.
      END DOCUMENTATION;
   END SUBSYSTEM CLASS;
```

```
[card_reader]: SUBSYSTEM CLASS;
   request_in: INPORT;
      BUFFER SUBCOMPONENTS; read_request OF [read_request_message]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   card_out: OUT PORT;
      BUFFER SUBCOMPONENTS; card_image OF [card]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   reader: CONTROL PROCESS;
      MODEL; ITERATE; RECEIVE request in;
                      SET card_image TO defined;
                      SEND card_out;
                END ITERATE;
         END MODEL;
      END CONTROL PROCESS;
   END SUBSYSTEM CLASS;
```

```
[printer]: SUBSYSTEM CLASS;
   DOCUMENTATION; The model is of a printer which receives
      carriage control signals separately from lines to be
      printed.  It is not necessary that each carriage
      control signal be followed by a line to be printed.
      END DOCUMENTATION;
   carriage_control: IN PORT;
      BUFFER SUBCOMPONENTS; signal OF [carriage_control_signal]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   contents: IN PORT;
      BUFFER SUBCOMPONENTS; line OF [print_line]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   print: CONTROL PROCESS;
      MODEL; ITERATE; RECEIVE carriage_control;
                        MAYBE; RECEIVE contents;
                           END MAYBE;
                END ITERATE;
         END MODEL;
      END CONTROL PROCESS;
   END SUBSYSTEM CLASS;
```

In giving these models, we have indicated the need to have several different types of information. In some cases, we have indicated "values" that pieces of information should assume. This can be recorded more explicitly by using DDN monitor classes to record the names for these types of information and their possible "values" as delineated so far.

```
[real_address]: MONITOR CLASS;
   END MONITOR CLASS;

[program_message]: MONITOR CLASS;
   DOCUMENTATION; messages sent by a program to the operator
      through the operator's console;
      END DOCUMENTATION;
   END MONITOR CLASS;

[operator_reply]: MONITOR CLASS;
   STATE SUBSETS; ans END STATE SUBSETS;
   END MONITOR CLASS;

[read_request_message]: MONITOR CLASS;
   END MONITOR CLASS;
```

```
[card]: MONITOR CLASS;
   STATE SUBSETS; defined END STATE SUBSETS;
   END MONITOR CLASS;

[carriage_control_signal]: MONITOR CLASS;
   END MONITOR CLASS;

[print_line]: MONITOR CLASS;
   END MONITOR CLASS;
```

Before giving the operational parts of the operating system itself, another primitive part of the overall system that needs to be modelled is a program running under the operating system. The system is a multiprogrammed one, so the class definition facilities are used to model a generic program running under the operating system. With respect to the devices and resources managed by the operating system, the important thing to describe about a program is that it produces a nondeterministic sequence of uses of the various resources and devices.

```
[program]: SUBSYSTEM CLASS;
   card_request: OUT PORT;
      BUFFER SUBCOMPONENTS; read_request OF [read_request_message]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   read_card: IN PORT;
      BUFFER SUBCOMPONENTS; card_image OF [card]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   printer_control: OUT PORT;
      BUFFER SUBCOMPONENTS: signal OF [carriage_control_signal]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   line_contents: OUT PORT;
      BUFFER SUBCOMPONENTS; line OF [print_line]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   to_operator: OUT PORT;
      BUFFER SUBCOMPONENTS; message OF [program_message]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   from_operator: IN PORT;
      BUFFER SUBCOMPONENTS; reply OF [operator_reply]
         END BUFFER SUBCOMPONENTS;
      END INPORT;
   access: OUT PORT;
      BUFFER SUBCOMPONENTS; address OF [virtual_address]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   END SUBSYSTEM CLASS;

[virtual_address]: MONITOR CLASS;
   STATE SUBSETS; defined END STATE SUBSETS;
   END MONITOR CLASS;
```

```
'[program]: SUBSYSTEM CLASS' run: CONTROL PROCESS;
   MODEL; ITERATE; SELECT;
                      (PERHAPS) COMMENT read operation;
                                SET read_request TO defined;
                                SEND card_request;
                                RECEIVE read_card;
                      (PERHAPS) COMMENT write operation;
                                SET signal TO defined;
                                SEND printer_control;
                                MAYBE; SET line TO defined;
                                       SEND line_contents;
                                   END MAYBE;
                      (PERHAPS) COMMENT interact with operator;
                                SET message TO defined;
                                SEND to_operator;
                                MAYBE; COMMENT not all messages
                                               require a reply;
                                       RECEIVE from_operator;
                                   END MAYBE;
                      (OTHERWISE) COMMENT read or write a location
                                          in program's virtual
                                          memory space;
                                SET address TO defined;
                                SEND access;
                      END SELECT;
             END ITERATE;
      END MODEL;
   END CONTROL PROCESS;
```

Notice that there is an inconsistency among these models — the operator always replies to each message that a program sends whereas the program does not always expect a reply. This could be uncovered by simulation-based testing at this level of modelling or by more formal analysis. This inconsistency will be removed when the operation of the system with respect to conversations between a program and the operator is elaborated later.

The description of the operating system at this level of modelling may be completed by giving models of the major operational parts of the operating system. First, the address translator which converts addresses in the virtual memory spaces of the programs into the actual memory space of the memory system:

```
[address_translator]: SUBSYSTEM CLASS;
   virtual_space: IN PORT;
      BUFFER SUBCOMPONENTS; v_address OF [virtual_address]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   actual_space: OUT PORT;
      BUFFER SUBCOMPONENTS; a_address OF [real_address]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   END SUBSYSTEM CLASS;
```

```
'[address_translator]: SUBSYSTEM CLASS' translate: CONTROL PROCESS;
   MODEL; ITERATE; RECEIVE virtual_space;
                   SET a_address TO defined;
                   SEND actual_space;
             END ITERATE;
      END MODEL;
   END CONTROL PROCESS;
```

Before giving models of the other major processing portions of the operating systems, the definitions of the pieces of information processed by the system should be updated.

```
'[read_request_message]: MONITOR CLASS'
   STATE SUBSETS defined END STATE SUBSETS;

'[carriage_control_signal]: MONITOR CLASS'
   STATE SUBSETS defined END STATE SUBSETS;

'[print_line]: MONITOR CLASS'
   STATE SUBSETS defined END STATE SUBSETS;

'[program_message]: MONITOR CLASS'
   STATE SUBSETS defined END STATE SUBSETS;

'[real_address]: MONITOR CLASS'
   STATE SUBSETS defined END STATE SUBSETS;
```

At this level of modelling, there are no further distinctions that can be made among the "values" of these pieces of information.

The next processing module within the operating system is the handler of messages between the programs and the operator. At this level of modelling, the handler is either passing on a program's message, perhaps with some accesses to the handler's virtual memory space, or passing on the operator's reply, again possibly with some accesses to the handler's virtual memory space. The description of this class is parameterized with respect to the number of programs that can be handled.

```
[message_interpreter]: SUBSYSTEM CLASS;
   QUALIFIERS: #_of_programs END QUALIFIERS;
   message_in: ARRAY [1::#_of_programs] OF IN PORT;
      BUFFER SUBCOMPONENTS; message OF  [program_message]
         END BUFFER SUBCOMPONENTS;
      END INPORT;
   message_out: OUT PORT;
      BUFFER SUBCOMPONENTS; message_to_operator OF [program_message]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   reply_in: IN PORT;
      BUFFER SUBCOMPONENTS; reply OF [operator_reply]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   reply_out: ARRAY[1::#_of_programs] OF OUT PORT;
      BUFFER SUBCOMPONENTS; reply_to_program OF [operator_reply]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   END SUBSYSTEM CLASS;

[message_interpreter]: SUBSYSTEM CLASS;
   access: OUT PORT;
      BUFFER SUBCOMPONENTS; address OF [virtual_address]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   collector: ARRAY[1::#_of_programs] OF CONTROL PROCESS;
      DOCUMENTATION; These control processes serve to funnel all
         of the messages from all of the programs into one stream
         and therefore to model that the message handler can
         handle the messages in any order.
         END DOCUMENTATION;
      message_stream: LOCAL OUT PORT;
         BUFFER SUBCOMPONENTS; hold_message OF [program_message]
            END BUFFER SUBCOMPONENTS
         END LOCAL OUT PORT;
      MODEL: ITERATE; RECEIVE message_in(MY_INDEX);
```

```
                   SET hold_message(MY INDEX) TO message(MY_INDEX);
                   SEND message_stream(MY_INDEX);
             END ITERATE;
      END MODEL;
   END CONTROL PROCESS;
handler: CONTROL PROCESS;
   get_message: LOCAL IN PORT;
      BUFFER SUBCOMPONENTS; one_of_the_messages OF [program message]
         END BUFFER SUBCOMPONENTS;
      END LOCAL IN PORT;
   MODEL; ITERATE; RECEIVE get_message;
                   ITERATE PERHAPS;
                      SET address TO defined;
                      SEND access;
                      END ITERATE;
                   SET message_to_operator TO defined;
                   SEND message_out;
                   MAYBE; COMMENT reply not always expected;
                      RECEIVE reply_in;
                      ITERATE PERHAPS;
                         SET address TO defined;
                         SEND access;
                         END ITERATE;
                      FOR SOME i IN [1::#_of_programs];
                         SET reply_to_program(i) TO defined;
                         SEND reply_out(i);
                         END FOR;
                      END MAYBE;
             END ITERATE;
      END MODEL;
   END CONTROL PROCESS;
CONNECTIONS;
   FOR ALL i IN [1::#_of_programs];
      PLUG(collector(i)|message_stream, handler|get_message);
      END FOR;
   END CONNECTIONS;
END SUBSYSTEM CLASS;
```

The connections among the ports of the control processes serve, as the comment in the documentation of the collector control processes indicates, to set up a message communication network which funnels all of the messages into one stream. This network, when there are four programs, may be graphically represented as in Figure 2.

A further elaboration of the message interpreter, given later, will indicate that an alternative communication network is really used. This one, and the models of the control processes, serve to give an abstract description of the interactions of this part of the operating system with the other parts of the operating system.

The remaining major part of the operating system is the spooling subsystem. In the following model of this part, we exhibit an alternative to funnelling message from many sources into one stream — the spooler is set up to poll the various sources of requests in some nondeterministic (at this point anyway) order.

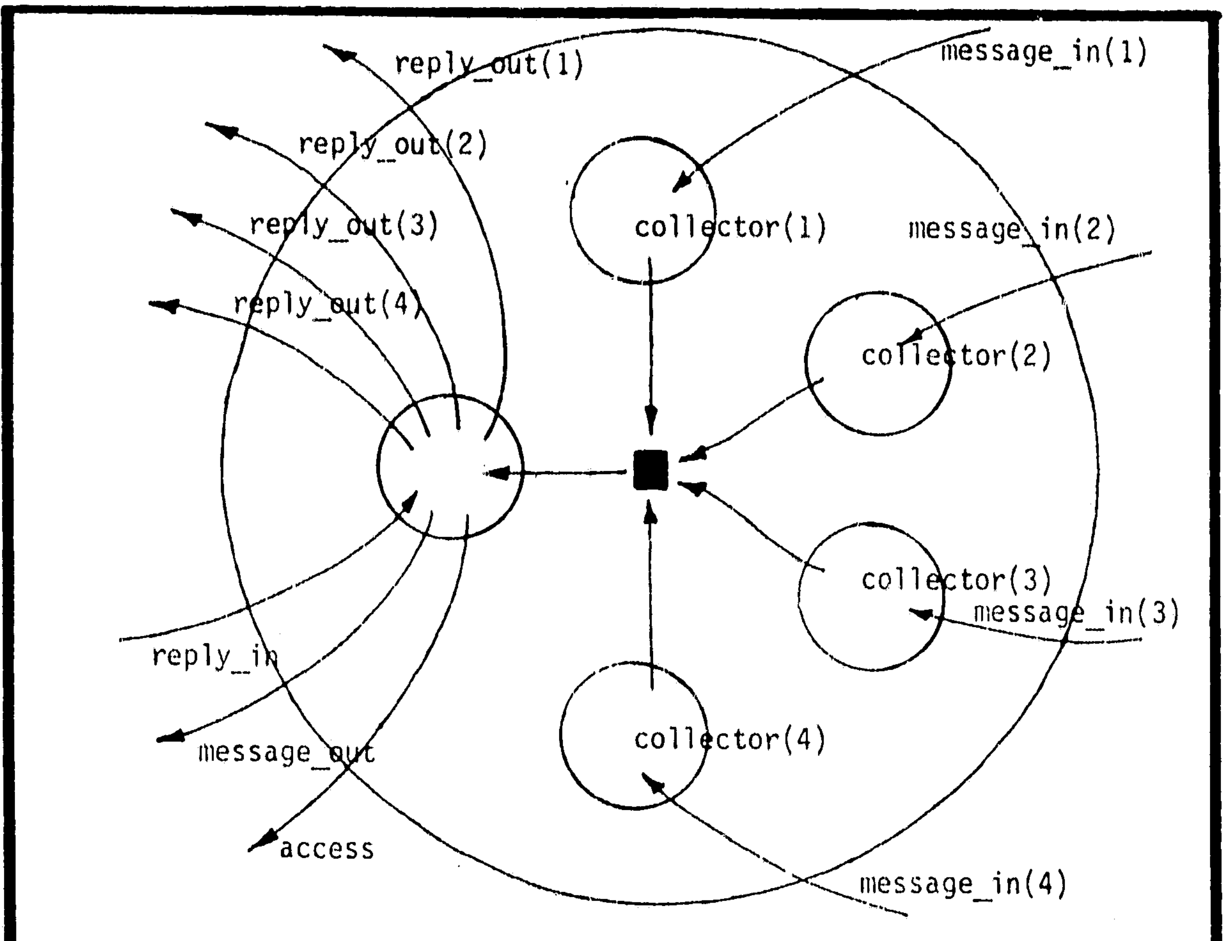

FIGURE 2

```
[spooling_system]: SUBSYSTEM CLASS;
   access: OUT PORT;
      BUFFER SUBCOMPONENTS; address OF [virtual_address]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   to_operator: OUT PORT;
      BUFFER SUBCOMPONENTS; message OF [program_message]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   from_operator: IN PORT;
      BUFFER SUBCOMPONENTS; reply OF [operator_reply]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   END SUBSYSTEM CLASS;

[spooling_system]: SUBSYSTEM CLASS;
   QUALIFIERS; #_of_user_programs END QUALIFIERS;
   read_request_in: ARRAY[1::#_of_user_programs] OF IN PORT;
      BUFFER SUBCOMPONENTS; program_read_request
                                   OF [read_request_message]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   read_request_out: OUT PORT;
      BUFFER SUBCOMPONENTS; read_request OF [read_request_message]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   read_card: IN PORT;
      BUFFER SUBCOMPONENTS: card_image OF [card]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   [illegible] ARRAY[illegible] OF OUT PORT;
   BUFFER SUBCOMPONENTS; delivered_card_image OF [card]
      END BUFFER SUBCOMPONENTS;
   END OUT PORT;
printer_control_in: ARRAY[1::#_of_user_programs] OF IN PORT;
   BUFFER SUBCOMPONENTS: signal OF [carriage_control_signal]
      END BUFFER SUBCOMPONENTS;
   END IN PORT;
printer_control_out: OUT PORT;
   BUFFER SUBCOMPONENTS; signal_to_printer
                               OF [carriage_control_signal]
      END BUFFER SUBCOMPONENTS;
   END OUT PORT;
line_contents_in: ARRAY[1::#_of_user_programs] OF IN PORT;
   BUFFER SUBCOMPONENTS; line OF [print_line]
      END BUFFER SUBCOMPONENTS;
   END IN PORT;
line_contents_out: OUT PORT;
   BUFFER SUBCOMPONENTS; line_to_printer OF [print_line]
      END BUFFER SUBCOMPONENTS;
   END OUT PORT;
spooler: CONTROL PROCESS;
   MODEL; ITERATE; SELECT;
                      (PERHAPS) COMMENT send a message to the
                                        operator;
                                SET message TO defined;
                                SEND to_operator;
                                MAYBE; RECEIVE from_operator;
                                   END MAYBE;
                      (PERHAPS) COMMENT read or write a location
                                        in virtual memory space;
                                SET address TO defined;
                                SEND access;
                      (OTHERWISE) COMMENT service a request
                                          from one of the user
                                          programs;
                          FOR SOME i IN [1::#_of_user_programs];
                             MAYBE; RECEIVE read_request_in(i);
                                    SET read_request TO defined;
                                    SEND read_request_out;
                                    RECEIVE read_card;
                                    SET delivered_card_image(i)
                                          TO card_image;
                                    SEND read_card_out(i);
                                ELSE:
                                    RECEIVE printer_control_in(i);
                                    SET signal_to_printer
                                          TO defined;
                                    SEND printer_control_out;
                                    MAYBE;
                                       RECEIVE line_contents_in(i);
                                       SET line_to_printer
                                             TO line(i);
                                       SEND line_contents_out;
                                       END MAYBE;
                                END MAYBE;
                      END SELECT;
             END ITERATE;
      END MODEL;
   END CONTROL PROCESS;
END SUBSYSTEM CLASS;
```

This concludes the description of all of the components for the operating system. The remaining step at this level of modelling is to describe the operating system itself, indicating its componentry and the network of communication pathways among the components.

```
[operating system]: SUBSYSTEM CLASS;
   QUALIFIERS;#_of_user_programs END QUALIFIERS;
   SUBCOMPONENTS;
      programs ARRAY[1::#_of_user_programs] OF [program],
      memory system OF [memory_control_system],
      console OF [operator_console],
      reader OF [card_reader],
      hard_copy OF [printer],
      translator OF [address_translator],
      interpreter OF [message_interpreter(#_of_user_programs+1)],
      spool OF [spooling_system(#_of_user_programs)]
      END SUBCOMPONENTS;
   CONNECTIONS;
      PLUG (translator|actual_space, memory_system|channel);
      PLUG (interpreter|message_out, console|message_in);
      PLUG (console|reply_out, interpreter|reply_in);
      PLUG (interpreter|access, translator|virtual_space);
      PLUG (spool|read_request_out, reader|request_in);
      PLUG (reader|card_out, spool|read_card);
      PLUG (spool|printer_control_out, hard_copy|carriage_control);
      PLUG (spool|line_contents_out, hard_copy|contents);
      PLUG (spool|access, translator|virtual_space);
      PLUG (spool|to_operator, interpreter|message_in(1));
      PLUG (interpreter|reply_out(1), spool|from_operator);
      FOR ALL i IN [1::#_of_user_programs];
         PLUG (program(i)|card_request, spool|read_request_in(i));
         PLUG (spool|read_card_out(i), program(i)|read_card);
         PLUG (program(i)|printer_control, spool|printer_control_in(i));
         PLUG (program(i)|line_contents, spool|line_contents_in(i));
         PLUG (program(i)|to_operator, interpreter|message_in(i+1));
         PLUG (interpreter|reply_out(i+1), program(i)|from operator);
         PLUG (program(i)|access, translator|virtual_space);
         END FOR;
      END CONNECTIONS;
   END SUBSYSTEM CLASS;
```

The communication network that is set up by this description is essentially that given in Figure 1 in [Riddle 72].

****

In [Riddle 72] the level of modelling for the operating system is elaborated for each of the major operational parts of the operating system. For purposes of example, we will carry that elaboration out for the message_interpreter and address_translator parts of the operating system. The elaboration of the spooling_system, using DDN, is left as an exercise.

For the address_translator, we need first to model the external storage that is used to hold paged-out portions of the virtual spaces.

```
[external_storage]: SUBSYSTEM CLASS;
   read_request: IN PORT;
      BUFFER SUBCOMPONENTS; request OF [io_request]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   write_request: IN POR
      BUFFER SUBCOMPONEN      rite_request OF [io_request]
           BUFFER SUBCOMPONENTS;
         END BUFFER CONDITIONS;
      END IN PORT;
   read_done: OUT PORT;
      BUFFER SUBCOMPONENTS; signal OF [io_done_signal]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   write_done: OUT PORT;
      BUFFER SUBCOMPONENTS; write_signal OF [io_done_signal]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   io: CONTROL PROCESS;
      MODEL; ITERATE; MAYBE;
                         RECEIVE read_request;
                         SET signal TO accomplished;
                         SEND read_done;
                       ELSE;
                         RECEIVE write_request;
                         SET write_signal TO accomplished;
                         SEND write_done;
                       END MAYBE;
                END ITERATE;
         END MODEL;
      END CONTROL PROCESS;
   END SUBSYSTEM CLASS;

[io_request]: MONITOR CLASS;
   STATE SUBSETS; read, write END STATE SUBSETS;
   END MONITOR CLASS;

[io_done_signal]: MONITOR CLASS;
   STATE SUBSETS; accomplished END STATE SUBSETS;
   END MONITOR CLASS;
```

We have used the buffer conditions construct of DDN to indicate that only write requests may come in through the write_request port.

We also need a semaphore.

```
[semaphore]: MONITOR CLASS;
   STATE SUBSETS; uninitialized, zero, one
      END STATE SUBSETS;
   DOCUMENTATION; This is a binary semaphore which may be used
      for mutual exclusion.  The operation of the procedures is
      not elaborated here — they may be "programmed" using
      signals and waits upon condition variables.
      END DOCUMENTATION;
   initialize: PROCEDURE;
      TRANSITIONS; uninitialized → one
         END TRANSITIONS;
      END PROCEDURE;
   p: PROCEDURE;
      TRANSITIONS; one → zero; zero → zero
         END TRANSITIONS;
      END PROCEDURE;
   v: PROCEDURE;
      TRANSITIONS; zero → one
         END TRANSITIONS;
      END PROCEDURE;
   END MONITOR CLASS;
```

Now we can specify the parts of the address translator as control processes.

```
[address_translator]: SUBSYSTEM CLASS;
   virtual_space: INPORT;
      BUFFER SUBCOMPONENTS; v_address OF [virtual_address]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   actual_space: OUT PORT;
      BUFFER SUBCOMPONENTS; a_address OF [real_address]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   SUBCOMPONENTS;
      storage OF [external_storage],
      mutex OF [semaphore]
      END SUBCOMPONENTS
   check: CONTROL PROCESS;
      set_up_swap: LOCAL OUT PORT;
         BUFFER SUBCOMPONENTS; signal OF [activate_signal]
            END BUFFER SUBCOMPONENTS;
         END LOCAL OUT PORT;
      BODY; ITERATE; RECEIVE virtual_space;
                     mutex.p;
                     COMMENT access the page tables;
                     mutex.v;
                     IF PERHAPS
                        THEN a_address.assign;
                             SEND actual_space;
                        ELSE signal.assign;
                             SEND set_up_swap;
                        END IF;
                  END ITERATE;
         END BODY;  END CONTROL PROCESS;  END SUBSYSTEM CLASS;
```

In this control process definition, we have used a body textual unit to indicate that what is being defined is an actual part of the subsystem rather than just a model. Within the body, we have used nondeterminism to indicate that it is not yet completely specified as to how the decision is made as to whether the page is in the actual memory system or on external storage. This is not really legal in DDN — a legal description would require the definition of a flag variable that was set (nondeterministically) to either true or false within the critical section and was used to control the subsequent flow of processing.

The definition of the address_translator is completed with the following textual units.

```
[address_translator]: SUBSYSTEM CLASS;
   swap: CONTROL PROCESS;
      wait_for_activation: LOCAL IN PORT;
         BUFFER SUBCOMPONENTS; signal OF [activate_signal]
            END BUFFER SUBCOMPONENTS;
         END LOCAL IN PORT;
      read_request: LOCAL OUT PORT;
         BUFFER SUBCOMPONENTS; request OF [io_request]
            END BUFFER SUBCOMPONENTS;
         END LOCAL OUT PORT;
      read_done: LOCAL IN PORT;
         BUFFER SUBCOMPONENTS; signal OF [io_done_signal]
            END BUFFER SUBCOMPONENTS;
         END LOCAL IN PORT;
      write_request: LOCAL OUT PORT;
         BUFFER SUBCOMPONENTS; write_request OF [io_request]
            END BUFFER SUBCOMPONENTS;
         END LOCAL OUT
      ite_done: LOCAL IN PORT;
                                                 [io_done_signal]
            END BUFFER SUBCOMPONENTS;
         END LOCAL IN PORT;
      BODY; ITERATE; RECEIVE wait_for_activation;
                     mutex.p: COMMENT check for page unchanged;
                     mutex.v;
                     IF PERHAPS
                        THEN write_request.assign;
                             SEND write_request;
                             RECEIVE write_done;
                        END IF;
                     request.assign;
                     SEND read_request;
                     RECEIVE read_done;
                     mutex.p; COMMENT update the page tables;
                     mutex.v;
                     a_address.assign;
                     SEND actual_space;
                     END ITERATE;
         END BODY;
      END CONTROL PROCESS;
   END SUBSYSTEM CLASS;

[address_translator]: SUBSYSTEM CLASS;
   CONNECTIONS;
      PLUG (check|set_up_swap, swap|wait_for_activation);
      PLUG (swap|read_request, storage|read_request);
      PLUG (storage|read_done, swap|read_done);
      PLUG (swap|write_request, storage|write_request);
      PLUG (storage|write_done, swap|write_done);
      END CONNECTIONS;
```

The graphical representation of this connection network is essentially that appearing as Figure 6 in [Riddle 72]. The major difference is that in the Figure in [Riddle 72], the semaphore is represented as a link process because the precursor of DDN did not have the concept of monitor class.

This elaboration indicates one of the major purposes of control processes in addition to their role in modelling. When components within a subsystem are particular to that subsystem, a body for a control process may be prepared to indicate the algorithm for the message handling carried out by the component. DDN would allow a class definition to be prepared and then an instance of that class to be declared as a subcomponent within the subsystem -- as was done for the external storage component. But it is often sufficient to indicate the component directly as a control process body, and this serves the additional purpose of drawing a direct correspondence between the subsystem's modelled behavior and the operation of the subsystem's components.

In these textual units, we have again used procedures named assign. We should define these within the class definitions, but this is not really very illustrative, so we will skip that.

For the elaboration of the message_interpreter, we will not only specify the operation of some of its components, but will also elaborate the description with respect to the types of messages that are processed. The intent of this elaboration is to specify some aspects of conversations between a program and the operator. First, we elaborate the types of messages that can be sent by the programs to the operator.

```
[program_message]: MONITOR CLASS;
   STATE SUBSETS; write, write_and_wait, terminate
      END STATE SUBSETS;
   set_to_terminate: PROCEDURE;
      TRANSITIONS;     → terminate
         END TRANSITIONS;
      END PROCEDURE;
   END MONITOR CLASS;
```

We have included the procedure set_to_terminate because it will be needed in later descriptions. The state subsets indicate that the program may send a message to the operator without a reply expected (write), send a message with a reply expected (write_and_wait), or indicate that the conversation with the operator may be terminated.

In the elaboration of the message_interpreter, it will attach an identification of the originating program to the message before passing it on to the operator. Thus we need a class definition that describes these coded messages.

```
[encoded_message]: MONITOR CLASS;
   QUALIFIERS; #_of_programs END QUALIFIERS;
   STATE VARIABLES;
      id: VALUES (1::#_of_programs),
      content: VALUES (message, null)
      END STATE VARIABLES;
   STATE SUBSETS;
      write: <<--, content=message>>,
      write_and_wait: <<--, content=message>>,
      terminate: <<--, content=null>>
      END STATE SUBSETS;
   assign: PROCEDURE;
      PARAMETERS:
         id VALUE OF [1::#_of_programs],
         message_to_be_sent VALUE OF [program_message]
         END PARAMETERS;
      TRANSITIONS; message_to_be_sent=write →write,
                   message_to_be_sent=write_and_wait → write_ and_ wait,
                   message_to_be_sent=terminate → terminate
         END TRANSITIONS;
      END PROCEDURE;
   END MONITOR CLASS;
```

We also need similar class definitions for the messages sent from the operator to the program.

```
[operator_reply]: MONITOR CLASS;
   STATE SUBSETS; ans, suspend
      END STATE SUBSETS;
   assign: PROCEDURE;
      PARAMETERS;
         reply_to_be_sent VALUE OF [encoded_reply]
         END PARAMETERS;
      TRANSITIONS; reply_to_be_sent=ans → and
         END TRANSITIONS;
      DOCUMENTATION;  This procedure may not be legally invoked
         when the operator has sent a suspend message.
         END DOCUMENTATION;
      END PROCEDURE;
   END MONITOR CLASS;
```

```
[encoded_reply]: MONITOR CLASS;
   QUALIFIERS;#_of_programs END QUALIFIERS;
   STATE VARIABLES;
      id: VALUES (1::#_of_programs),
      content: VALUES (message, null)
      END STATE VARIABLES;
   STATE SUBSETS;
      ans: <<--, content=message>>,
      suspend: <<--, content_null>>
      END STATE SUBSETS;
   find_id: PROCEDURE;
      PARAMETERS;
         id RESULT OF [1::#_of_programs]
         END PARAMETERS;
   END MONITOR CLASS;
```

With these monitor classes, we are in a position to give the elaboration of the message_interpreter.

```
[message_interpreter]: SUBSYSYEM CLASS;
   QUALIFIERS; #_of_programs END QUALIFIERS;
   message_in: ARRAY[1::#_of_programs] OF INPORT;
      BUFFER SUBCOMPONENTS; message OF [program_message]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   message_out: OUT PORT;
      BUFFER SUBCOMPONENTS; message_to_operator
                                  OF [encoded_message(#_of_programs)]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   reply_in: INPORT;
      BUFFER SUBCOMPONENTS; reply OF [encoded_reply(#_of_programs)]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   reply_out: ARRAY[1::#_of_programs] OF OUT PORT;
      BUFFER SUBCOMPONENTS; reply_to_program OF [operator_reply]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   access: OUT PORT;
      BUFFER SUBCOMPONENTS: address OF [virtual_address]
         END BUFFER SUBCOMPONENTS:
      END OUT PORT;
   SUBCOMPONENTS;
      mutex OF [semaphore]
      END SUBCOMPONENTS;
   transfer: ARRAY[1::#_of_programs] OF CONTROL PROCESS;
      pass_message_in: LOCAL OUT PORT;
         BUFFER SUBCOMPONENTS; passed_message OF [program_message]
            END BUFFER SUBCOMPONENTS;
         END LOCAL OUT PORT;
      MODEL; ITERATE; RECEIVE message_in(MY_INDEX);
                      SET passed_message(MY_INDEX) TO message(MY_INDEX);
                      SEND pass_message_in(MY_INDEX);
                END ITERATE;
         END MODEL;
      END CONTROL PROCESS;
   encode: ARRAY[1::#_of_programs] OF CONTROL PROCESS;
      get_message_in: LOCAL IN PORT;
         BUFFER SUBCOMPONENTS; message_from_program_or_decode
                                        OF [program_message]
            END BUFFER SUBCOMPONENTS;
         END LOCAL IN PORT;
      BODY; ITERATE;
```

```
          RECEIVE get_message_in(MY_INDEX);
          mutex.p;
          WHILE message_from_program_or_decode(MY_INDEX)
                      = write_and_wait;
             ITERATE PERHAPS;
                address.assign;
                SEND access;
                END ITERATE;
             message_to_operator.assign(MY_INDEX,
                      message_from_program_or_decode(MY_INDEX));
             SEND message_out;
             RECEIVE get_message_in(MY_INDEX);
             END WHILE;
             IF message_from_program_or_decode(MY_INDEX)
                   = write;
                THEN; ITERATE PERHAPS;
                         address.assign;
                         SEND access;
                         END ITERATE;
                      message_to_operator.assign(MY_INDEX,
                        message_from_program_or_decode(MY_INDEX));
                      SEND message_out;
                END IF;
             mutex.v;
             END ITERATE;
      END BODY;
   END CONTROL PROCESS;
decode: CONTROL PROCESS;
   transfer_suspend: ARRAY[1::#_of_programs] OF LOCAL OUT PORT;
      BUFFER SUBCOMPONENTS: term_message OF [program_message]
         BUFFER SUBCOMPONENTS;
      END LOCAL OUT PORT;
   LOCAL SUBCOMPONENTS;
      id OF [1::#_of_programs]
      END LOCAL SUBCOMPONENTS;
   BODY;  ITERATE;
             RECEIVE reply_in;
             reply.find_id(id);
             IF reply = suspend
                THEN; term_message(id).set_to_terminate;
                      SEND transfer_suspend(id);
                ELSE; reply_to_program(id).assign(reply);
                      SEND reply_out(id);
                END IF;
             END ITERATE;
      END BODY;
   END CONTROL PROCESS;
CONNECTIONS;
   FOR ALL i IN [1::#_of_programs];
      PLUG (transfer(i)|pass_message_in,
                      encode(i)|get_message_in);
      PLUG (decode|transfer_suspend(i),
                      encode(i)|get_message_in);
      END FOR;
   END CONNECTIONS
END SUBSYSTEM CLASS;
```

This completes the elaboration of the message_interpreter. It remains to update the model of the operator's console to reflect the new level of modelling achieved in the model of the message_interpreter.

```
[operator_console]: SUBSYSTEM CLASS;
   DOCUMENTATION; This models the console and the operator
      using it.
      END DOCUMENTATION;
   message_in: IN PORT;
      BUFFER SUBCOMPONENTS; message OF [encoded_message(#_of_programs)]
         END BUFFER SUBCOMPONENTS;
      END IN PORT;
   END SUBSYSTEM CLASS;

[operator_console]: SUBSYSTEM CLASS;
   reply_out: OUT PORT;
      BUFFER SUBCOMPONENTS; reply OF [encoded_reply(#_of_programs)]
         END BUFFER SUBCOMPONENTS;
      END OUT PORT;
   operator: CONTROL PROCESS;
      LOCAL SUBCOMPONENTS; id OF [id::#_of_programs]
         END LOCAL SUBCOMPONENTS;
      BODY; ITERATE;
               RECEIVE message_in;
               message.find_id(id);
               IF message = write_and_wait;
                 THEN; reply.set_id(id);
                       reply.set_type;
                       SEND reply_out;
                       IF reply = suspend AND PERHAPS;
                          THEN; reply.set_some_id;
                                reply.set_message;
                                SEND reply_out;
                          END IF;
                 END IF;
               END ITERATE;
         END BODY;
      END CONTROL PROCESS;
   QUALIFIERS; #_of_programs END QUALIFIERS;
   END SUBSYSTEM CLASS;
```

This description requres the definition of four procedures to operate upon encoded replies.

```
'[encoded_reply]: MONITOR CLASS' set_id: PROCEDURE;
   PARAMETERS;
      id VALUE OF [1::#_of_programs]
      END PARAMETERS;
   END PROCEDURE;

'[encoded_reply]: MONITOR CLASS' set_type: PROCEDURE;
   TRANSITIONS;  → ans,  → suspend
      END TRANSITIONS;
   END PROCEDURE;

'[encoded_reply]: MONITOR CLASS' set_some_id: PROCEDURE;
   END PROCEDURE;

'[encoded_reply]: MONITOR CLASS' set_message: PROCEDURE;
   TRANSITIONS;  → ans
      END TRANSITIONS;
   END PROCEDURE;
```

Note the use of nondeterminism in the procedure set_type. This makes the description of the operation of the operator_console be a nondeterministic one even though a body is given for the control process.