A description scheme to aid the design of collections of concurrent processes*

by WILLIAM E. RIDDLE
University of Colorado
Boulder, Colorado

JOHN H. SAYLER and ALAN R. SEGAL
University of Michigan
Ann Arbor, Michigan

ALLAN M. STAVELEY
New Mexico Institute of Mining and Technology
Socorro, New Mexico

and

JACK C. WILEDEN
University of Massachusetts
Amherst, Massachusetts

INTRODUCTION

Abstract data types have recently emerged as an important facility for the specification of sequential programs. Originally defined in the language SIMULA,1 abstract data types were soon recognized as useful in structured programming.2 Their recent refinement has taken three directions. First, they have been developed to support top-down design methods.3 Second, they have been extended so as to provide a basis for formal verification.4 Finally, they have been used as the basis for a tool to aid program development5 which admits rigorous, but perhaps incomplete, analysis.6 The variety of these uses indicates the breadth of the benefits which accrue from facilities for high-level, abstract description.

While abstract data types are convenient for the description of a software system's data storage components, they are not convenient for the succinct description of those system components which are more for the processing than the storage of data. This is particularly true when the components operate concurrently.** The major problem is that abstract data types are oriented towards describing components as structures of data which are operated upon via procedure calls. Many components—e.g., a text editor in an operating system or a file system in a multiprocessor computing facility—are not naturally described in this manner.

Extensions and modifications to abstract data types could be developed to increase their effectiveness in describing processing components in software systems—this has been done in the Gypsy system7 and the Modula programming language.8 In this paper, however, we develop a description scheme that retains many of the concepts of abstract data types but builds upon the concept of a sequential process. Further, the description scheme developed in this paper focuses upon the succinct modelling of a system's processing components rather than their detailed programming.

The modelling scheme presented here is an extension of a scheme for describing sequential process interaction9 that was developed as a basis for software system analysis.10 It has been prepared for use in the Design Realization, Evaluation And Modelling (DREAM) system11 which provides bookkeeping and analysis aid to the designers of large-scale software systems. The scheme is part of the DREAM Design Notation (DDN) which has been developed to permit the incremental development of a system's design as a series of design description fragments, called textual units.

In the next section, the basis for the modelling scheme is established by developing an abstract view of sequential process interaction as occurring through message interchange. The following sections then develop the model of
A subsystem, i.e., a collection of concurrent sequential processes. Subsystem interfaces and interaction are described first. Then a scheme for the non-procedural description of a subsystem's behavior is presented. The paper concludes with a brief discussion of the ways in which the modelling scheme may be beneficially used during the development of large-scale software systems.

The focus of this paper is upon DDN constructs for describing collections of concurrent sequential processes. More complete treatment of this material and discussions of other aspects of DREAM may be found in References 12 through 16. Also, the focus is upon the use of the constructs—their syntax is covered in Reference 17. Some justification for the constructs are given; others lie in the DREAM system's general philosophy which is discussed in Reference 18.

AN ABSTRACT VIEW OF SOFTWARE SYSTEMS

A software system may be viewed as composed of parts, subsystems, which operate concurrently and asynchronously and which interact either through shared data objects or by the sending and receiving of messages. This is a natural way in which to view multiprocessor systems, since these systems actually have components which interact by message transmission. But this view is also appropriate when it is merely a logical one and the system actually runs in a uniprocessor environment. In this case, the view facilitates the decomposition of the system (and thereby the mastery of its complexity) and message transmission is used only to model the actual interactions.

As an example of this view of software systems, consider the HEARSAY speech recognition system developed at Carnegie-Mellon University. One subsystem within the HEARSAY system is a data base, called the blackboard, which contains all the information about the utterance being recognized and the hypotheses which have been made as to its linguistic structure. The other subsystems within the HEARSAY system are processing components called knowledge sources. Each knowledge source inspects the information in the data base and augments or modifies it according to the rules which the knowledge source is programmed to enforce.

The interactions among these subsystems are as follows. Since it would be wasteful to have each knowledge source constantly inspect the data base to determine whether it should perform any processing, the data base is programmed to know which data base entries are of interest to each knowledge source and to send a signal to a knowledge source whenever one of the data base entries of interest to it changes value. Each knowledge source is reentrant and this (conceptually) means that each knowledge source has several internal servicer subsystems, each of which can perform the processing to be done by the knowledge source. When a knowledge source is signalled by the data base, one of these servicer subsystems is activated and it inspects the data base and makes any appropriate modifications. The interactions between a servicer and the data base consist of a sequence of message transmissions, with the data base returning messages in response to the requests sent by the servicer.

THE MODELLING SCHEME

A modelling scheme based upon this abstract view of software systems must have facilities for describing subsystems and their interactions. In this paper, we focus upon those subsystem interactions which occur through message transfer. We assume, therefore, the existence of a scheme for describing data objects that may be shared by a community of concurrent subsystems.* Such a scheme must be able to describe many aspects of shared data objects, but for the present discussion it is only important that the scheme allow the specification of the possible values of the data objects.

A subsystem model must describe three aspects of the subsystem. First, it must specify the interfaces to the subsystem not only in terms of the format of the information flowing through each interface but also in terms of what information may legally flow through each interface. Second, it must specify the legal sequences of message flow through the interfaces. Finally, it must relate the subsystem's operation at one point in time to its operation at a previous point in time—i.e., it must specify the more global aspects of the subsystem's operation.

In the following sections, each of these aspects is discussed and illustrated by examples which, taken together, comprise a description of the knowledge source subsystems of HEARSAY.** Since all of the knowledge source subsystems are similar with respect to their interactions with the data base, the description developed in the examples is of the class of knowledge source subsystems. To reflect that different instances of this class vary with respect to their details, such as the number of entries in the data base that are of interest to the knowledge source, we define a parameterized class. This means that part of the definition of the class is a specification of qualifiers which may be assigned values when an instance of the class is created. Class definitions and qualifiers are discussed more fully in Reference 12.

SUBSYSTEM INTERFACES

A subsystem's interface consists of a set of ports through which messages may flow. Conceptually, a port is a communication line along which messages flow, one at a time, and which does not have any storage capabilities. Ports correspond to uni-directional communication lines and therefore have a direction in or out.

---

* A scheme that was developed in conjunction with the work reported here is discussed in Reference 13.

** The description is an approximation of the actual structure and operation of the knowledge sources in HEARSAY. It reflects our understanding of the description which appears in Reference 20, but has also been constructed so as to provide examples of the facilities in DDN.
A subsystem's message flow characteristics may primarily be defined in terms of sequences of message transmissions through the subsystem's ports. This is analogous to defining a procedure in terms of a set of parameter/result pairs. More complex characteristics, called *global characteristics*, concern the correlations between transmissions occurring in different message transmission sequences. For example, in manipulating a stack, a pop operation must be preceded, at
At some point in time, by a push operation. In this section, the concern is with the more easily specified sequential message transmission characteristics. The specification of global message flow characteristics is treated in the next section.

Sequential message transmission characteristics are specified by giving a set of programs, each of which is a model of a sequential process, called a control process. Each control process model specifies a set of sequences of message flow through the subsystem. A control process is defined by a nondeterministic, procedural model which specifies the control process' message reception and transmission activity. Nondeterminism is allowed because it contributes to the clarity of the model, allowing succinct definition of the control process. A procedural specification is also used to enhance the clarity of the description.

Messages flow in and out through a subsystem's ports as a result of receive and send operations. For a send operation, a message is first composed using the values of the buffers associated with the port that is specified in the send operation. Then, the message is sent out through the port to be placed in the link to which the port is attached and thereby made available for reception by some subsystem. The control process which invoked the operation is suspended while the message is constructed and placed in the link. Since the links have infinite storage capacity, this suspension is relatively short and the send operation can therefore be considered to be a non-blocking operation.

When a receive operation is performed, the control process invoking the operation is suspended until a message is retrieved from the link to which the port specified in the receive operation is attached. The message is then decomposed and distributed among the buffers associated with the port. Since it is possible to request a message when none is available, the receive operation can block the control process which invoked it for relatively long periods.

Prior to a send operation, the subsystem will compute the values of the data objects which compose the message and place these values in the buffers. This computation may be lengthy and complicated, but neither its time consumption nor its detail are of interest in defining sequential message transmission characteristics. Thus, in a control process, computational detail is suppressed by modelling it with a set-to operation which may be applied to a buffer data object and causes the buffer to assume a value prescribed in the set-to operation.

To describe the sequential message transmission characteristics of [knowledge_source] subsystems, two control process types are needed. The first, specified in Figure 3, models the consumption of messages which arrive at the await port from the data base. This models the subsystem's operation with respect to signals sent to indicate a change of some data base entry of interest to the knowledge source. The variable MY_INDEX has a value in the range declared as the bounds of the array of control processes and is used to make each control process model distinct with respect to the buffers and ports to which it refers.

The second type of control process is specified in Figure 4. These control processes model the operation of the senders of the incoming signals, indicating that they present requests to the data base through one of the make_request ports and wait for answers to the requests through one of the get_answer ports. Note that nondeterminism may be specified in the set-to operation by giving a logical expression which specifies the set of values which could possibly be assigned to the data object as a result of the computation modelled by the set-to operation. The control process structure of a [knowledge_source] subsystem in which #_servers is 3 and #_values is 4 is graphically represented in Figure 5.

Each model is a control program over send and receive operations on ports and set-to operations upon buffers. The control constructs are Algol-like. There are constructs for definite iteration: an "ITERATE n TIMES" construct and a "FOR ALL i IN set_of_values" construct. WHILE and UNTIL constructs are available for indefinite iteration. Since many subsystems are designed to never terminate, there is also an ITERATE construct for infinite iteration. Conditional control may be specified by the usual forms of the IF construct. There is also a generalized CASE construct which allows "labelling" of the cases with logical expressions. All of the control constructs have a nondeterministic version. For example, the construct "ITERATE n OR MORE TIMES" may be used to indicate that the number

---

* Communication among asynchronous message transmitters and receivers requires a transmission controller that is able to store messages that have been sent but not yet received and requests for messages which have been lodged but not yet satisfied. In DDN, an idealized controller, called a link, is provided. Links hold messages and requests in (unbounded) bag data structures. Thus they do not necessarily pass messages on in the order the messages were sent, nor do they necessarily service requests for messages in the order the requests were lodged. Ports may be attached to links by a process described in Reference 14.

* The quoted prefix attached to the control process textual unit in Figure 3 indicates that this textual unit is intended to be an additional part of the definition of the [knowledge_source] subsystem class. This prefixing construction allows selective modification of a DREAM design description, thereby permitting incremental elaboration of a software system's design.
of times, while known before iteration begins, can be any number greater than or equal to \( n \).

**GLOBAL BEHAVIOR**

The facilities provided by a subsystem generally cannot be used in a totally arbitrary order. For example, the facility which a file system provides for opening a file must be used prior to the facilities for reading and writing the file. Thus there are correlations between what happens in one sequential message transmission and what may happen in another sequential message transmission.

This global behavior may be specified in terms of **events**, activities that may be observed external to the subsystem. Usually an event is the transmission of a message through a port, and thus may be identified as the execution of a send or receive operation in one of the control process models. In general, an event may be associated with the execution of any one of the instructions in a control process model. (Since control process models are used as part of the behavior description of a subsystem, they are known to an external observer.)

Global behavior is specified by defining a set of sequences of events. Note that this is what was accomplished procedurally via the control process models—each model defines a set of sequences of message transmissions where each sequence corresponds to an execution of the model. For the non-procedural specification of behavior, formal language theory provides several techniques since the set of events may be considered to be an alphabet and a behavior is then a language over this alphabet. 10

Two aspects of global behavior—one being required for correct operation and the other corresponding to a design decision—need to be specified for \([\text{knowledge_source}]\) subsystems. First, a servicer must not interact with the data base until after it has been activated by a signal indicating that an entry in the data base has changed. Second, the interactions with the data base must be ordered such that no request from any of the servicers is lodged until the previous request has been answered—the servicers must coordinate their interactions with the data base so that they utilize its facilities in a subroutine fashion.

To describe this behavior, we first define some events as indicated in Figure 6. The statement labels define names for events which correspond to the execution of the labelled statement. Note the use of the NULL instruction to allow denotation of the event "a servicer starts a sequence of interactions with the data base."

More macroscopic events are needed to conveniently define the global behavior. These are defined by the textual unit shown in Figure 7. The REPEAT operator denotes that the consult event may be repeated zero or more times. The SEQUENCE operator denotes that the events which are its arguments are sequenced in the specified order. Thus, the hear_and_do_something event corresponds to a signal ar-
riving from the data base and the subsequent interactions of a
servicer with the data base.

Using these events, the global behavior may be specified
as in Figure 8. The first part of the specification indicates
that hear...and...do...something events may proceed in par-
allel up to the limit imposed by having only #_servers ser-
vicers. The second part indicates that interactions with the
data base are mutually exclusive, i.e., ordered in time.

This example has used only a portion of the facilities
available in DDN for behavior specification. Events may be
associated with other aspects of a subsystem's operation,
more complex relationships among events may be defined,
relationships between events defined for different subsys-
tems may be established, and events which are not associ-
ated with any computational part of the system may be
defined. A complete description of DDN facilities for be-
behavior specification is given in Reference 15.

CONCLUSION

The modelling scheme presented in this paper allows the
abstract description of a collection of concurrent processes
(a subsystem). Models in this scheme describe the interac-
tions which may take place between a subsystem and other
subsystems. A model therefore provides a specification for
a subsystem which describes its behavior in relation to its
environment but hides the subsystem's operational detail.
A model consists of a definition of the subsystem's interface,
a procedural description of legal usage of the interface and
a non-procedural description of the legal usage of the sub-
ystem over time.

Models in the scheme are rigorous, unambiguous spec-
fications of the components within a software system. At
the very least, the models serve to record the facilities which
each component is to exhibit and the manner in which these
facilities are to relate. The models provide, therefore, spec-
fications which may be used to guide the eventual imple-
mentation of the components.

The models may also be used in the formulation of argu-
ments as to the appropriateness of a system's design. While
general techniques for formal verification cannot be defined,
both simulation and analytic techniques can be used to de-
rive information concerning the dynamic characteristics of
the interactions among the subsystems. The designer may
then use this information to determine whether or not the
specified desired behavior is actually achieved or to uncover
situations in which the desired behavior is not achieved.
This allows the designer to make corrections before pro-
ceeding and to proceed with increased confidence in the
validity of the design.

We have found the modelling scheme to be convenient for
the description of a variety of software systems. We feel
that it demarcates an important set of facilities which are
required for the specification of software systems and that
it will prove to be valuable as a basis for a variety of tools
to aid designers of large-scale software systems.

REFERENCES

1. Dahl, O. and K. Nygaard, "SIMULA—An Algol-Based Simulation Lan-
2. Hoare, C. A. R., "Notes on Data Structuring," in Dahl, Dijkstra and
3. Liskov, B. H. and S. N. Zilles, "Specification Techniques for Data Ab-
stractions," IEEE Trans. on Software Engineering, SE1, 1, March
1975, pp. 7-19.
tion in ALPHARD: Introduction to Language and Methodology," Report 76-
Laboratory, Univ. of Newcastle upon Tyne, England, September 1975.
1975 International Conf. on Reliable Software, Los Angeles, April 1975.
mentation of Verifiable Programs, ICSA-CMP-2, Certifiable Minicom-
puter Project, Univ. of Texas, Austin, January 1977.
8. Wirth, N., "Modula: A Language for Modular Multiprogramming," Soft-
ware—Practice and Experience, 7, 1977, pp. 3-33.
9. Riddle, W. E., "The Hierarchical Modelling of Operating System Struc-
Specification and Analysis," RSSM/25, Dept. of Computer and Com-
munication Sciences, University of Michigan, Ann Arbor, July 1976.
11. Riddle, W. E., J. Sayler, A. Segal and J. Wileden, "An Introduction to
the DREAM Software Design System," Software Engineering Notes, 2,
4, July 1977, pp. 11-23.
12. Riddle, W. E., "Hierarchical Description of Software System Structure,
RSSM/40, Department of Computer Science, Univ. of Colorado, Boulder,
October 1977.
Science, Univ. of Colorado, Boulder, in preparation.
Science, Univ. of Colorado, Boulder, November 1977.
15. Wileden, J., "Behavior Specification in a Software Design System," RSSM/43, Dept. of Computer and Communication Sciences, Univ. of
Michigan, Ann Arbor, in preparation.
16. Segal, A., "Design Description Management," RSSM/45, Dept. of Com-
puter and Communication Sciences, Univ. of Michigan, Ann Arbor, in
preparation.
Science, University of Colorado, Boulder, in preparation.
of Computer and Communication Sciences, Univ. of Michigan, in prepara-
tion.
106, December 1962, pp. 467-482. Also in Simon, Sciences of the Artifi-
20. Fennel, R., and V. R. Lesser, "Parallelism in Artificial Intelligence Prob-
lem Solving: A Case Study of Hearsay II," IEEE Trans. on Computers,
C-26, 2, February 1977.
22. Riddle, W. E., "A Formalism for the Comparison of Software Analysis
Techniques," RSSM/29, Dept. of Computer and Communication Sci-
ences, Univ. of Michigan, Ann Arbor, July 1977.
23. Cuny, J., "A DREAM Model of the RC 4000 Multiprogramming Sys-
tem," RSSM/46, Dept. of Computer and Communication Sciences, Univ.
of Michigan, Ann Arbor, July 1977.
Monitoring System," RSSM/49, Dept. of Computer and Communication
Sciences, Univ. of Michigan, Ann Arbor, July 1977.
25. Wileden, J., "DREAM Design Notation Example: Scheduler for a Mul-
tiprocessor System," RSSM/51, Dept. of Computer and Communication
Sciences, Univ. of Michigan, Ann Arbor, October 1977.
26. Segal, A. R., "DREAM Design Notation Example: A Multiprocessor
Supervisor," RSSM/53, Dept. of Computer and Communication Sci-
ences, Univ. of Michigan, Ann Arbor, August 1977.
27. Cuny, J., "The GM Terminal System," RSSM/63, Dept. of Computer and
Communication Sciences, Univ. of Michigan, Ann Arbor, August 1977.