HIERARCHICAL DESCRIPTION OF SOFTWARE SYSTEM ORGANIZATION

William E. Riddle
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-120-77 November, 1977

This work was supported by a grant from Sycor, Inc.

HIERARCHICAL DESCRIPTION

OF SOFTWARE SYSTEM ORGANIZATION

William E. Riddle
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309
303-492-7108

John H. Sayler
Computer and Communication Sciences Department
University of Michigan
Ann Arbor, Michigan 48105
313-764-8504

Alan R. Segal
Computer and Communication Sciences Department
University of Michigan
Ann Arbor, Michigan 48105
313-764-8504

Allan M. Stavely
Department of Computer Science
New Mexico Institute of Mining and Technology
Socorro, New Mexico 87801
505-835-5126

Jack C. Wileden
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003
413-545-0289

This work was supported by a grant from Sycor, Inc.

Abstract

Constructs for hierarchically describing the organization of large-scale software systems are presented. The constructs allow the explicit specification of subsystem sharing. Descriptions using these constructs are therefore not necessarily tree-like and hence are frequently more natural and clear. The constructs were developed as part of a language for describing a system's design, as opposed to its implementation. The efficacy of the constructs is argued with respect to this description task and the constructs are compared to similar ones available in programming languages.

Key Words and Phrases

hierarchical description, system organization, classes, instantiation control, DREAM, DDN

Introduction

Programming languages allow the description of a program's organization in terms of data structures, which form the program's passive data storage portions, and procedures, which form the program's active data manipulation portions. The trend in some recently developed programming languages, such as CLU [1], has been to unify the description of these different portions of a program through the use of abstract data types which allow the description of a collection of data objects along with the description of the procedures which operate upon them. One of the many justifications for incorporating abstract data types is that they support top-down design methods by allowing the hierarchical development of programs.

Abstract data types are frequently not convenient for describing the hierarchical organization of large-scale software systems, particularly when the systems are described as collections of asynchronous sequential processes. First, the emphasis upon describing the system's organization in terms of the organization of its data objects is sometimes inappropriate. A text editor, for example, is an organizational subunit in many operating systems but is not appropriately viewed as a data object. Second, abstract data types lead to a hierarchical organization which is tree-like and thus do not allow the clear description of organizations in which parts of the system are shared.

The emphasis of this paper is upon the second of these two problems. General constructs for describing the organization of large-scale software systems are developed. These constructs are an extension of ones developed for the TOPD program design system [2], which were themselves an amalgamation of the class concept of SIMULA [3] and some of the original work on abstract data types [4]. The constructs presented here differ significantly in providing a means of explicitly defining subcomponent sharing. Thus, the constructs lead to descriptions that are frequently more clear than the tree-like descriptions typical of other schemes based on user definition of classes of objects.

The constructs discussed here are part of the design language developed for the Design Realization Evaluation and Modelling (DREAM) system [5]. DREAM is a tool for the design of large-scale software systems, providing the designer with bookkeeping and analysis aid. A description in the DREAM Design Notation (DDN) consists of a collection of (nested) description fragments, called textual units. DREAM provides facilities which allow the user (or users, in the case of a design being carried out by a design team) to modify the information in a data base on a textual unit basis.

The focus of this paper is upon the DDN constructs for describing a system's organization. Other aspects of DDN are discussed in [6], [7], [8] and [9]. Also, the focus is upon the use of the constructs -- their syntax is covered in [10]. Some

justifications for the constructs are given; others lie within the DREAM system's general philosophy which is discussed in [11].

Hierarchical System Description

In DDN, a system is considered to be decomposable into interacting parts, called objects. This decomposition is hierarchical, with objects being composed of subobjects which are, in turn, composed of sub-subobjects, etc. Each object is an instance of some general class of objects and has the attributes specified in the class definition for that class. These attributes are each specified by a textual unit.

The textual unit of primary interest here is that which defines the subcomponents of an object in the class. Each subcomponent is an object or a (homogeneous) structure of objects. Part of a class definition is a declaration of the subcomponents of objects in the class. The declaration gives, for each subcomponent, a name and the class of which the subcomponent's objects are instances. If the subcomponents are not declared for a class, then objects of that class are primitive objects.

A DDN description of a system consists of a collection of class definitions. An instance of the system may be created by a process, called instantiation, which consists of two phases -- generation and configuration. Each step in the generation phase

consists of creating an instance of a non-primitive object by creating instances of all of the object's primitive subobjects and causing, at the next step in the generation phase, the creation of instances of all of the object's non-primitive subobjects. During the configuration phase, objects are equated to create new, shared objects.

In DDN, class definitions may not be recursive and the instantiation process is therefore guaranteed to terminate. Also, the instantiation process is carried out before the system is executed or interpreted and hence DDN may only be used to describe systems having a static organization.

A record of the instantiation process may be prepared by drawing a node for each object and a directed arc from node x to node y if the object corresponding to node y is a subobject within the object corresponding to node x. This record is in general a directed, acyclic graph called an instantiation graph. When there are no shared subobjects then the instantiation graph is a tree and is accordingly called an instantiation tree. If there is a path from a node x to a node y in the instantiation graph, then the object corresponding to node y is said to be a part of the object corresponding to the node x.

With this brief overview of the DDN approach to describing a system's hierarchical structure, we turn to a more complete description of the DDN constructs. Throughout the following sections, we will be developing, as an example, a description of

part of the Hearsay speech recognition system [12]. This is a multiprocessor system developed at Carnegie Mellon University. It contains a data base which holds information about the utterance being recognized and the hypotheses about its linguistic structure. The data base is available to each of several knowledge sources which operate in parallel. Each knowledge source inspects the information in the data base and modifies it according to the rules of speech understanding that it is programmed to enforce. The data base activates a predefined set of knowledge sources whenever some data base entry of interest to these knowledge sources changes value. In the example, we focus upon describing the organizational attributes of the knowledge source (sub)systems.

Class Definitions

A class is defined in DDN by giving a textual unit for the class and several textual units nested within it which define the attributes of objects in the class. For example, suppose that we have a class, called [knowledge_source][1], of (sub)systems which asynchronously manipulate information in a shared data base. A textual unit which defines the existence of this class, but does not give any information about its attributes, would be:

---

[1] It is a convention in DDN to enclose an identifier in square brackets when it is used to name a class.

```
[knowledge_source]: SUBSYSTEM CLASS;
  END SUBSYSTEM CLASS;
```

The textual units defining the attributes of objects in this class would appear nested within the textual unit.

In DDN, there are three types of classes, two of which are of concern in this paper. Monitor classes [6] define objects which correspond to shared data items. Monitor objects have a finite set of possible values and a built-in synchronization mechanism which sequentializes the operations done upon the objects. Subsystem classes [7] define objects which may operate asynchronously and concurrently, interacting with each other through message exchange and shared monitor objects. In this paper, we treat those textual units which are common to definitions for both these types of classes.

Qualified Class Definitions

Frequently, several class definitions will be quite similar, differing only with respect to minor details. One example is the collection of classes of bounded ranges of integers which differ with respect to the values which bound the range. Another example is the collection of classes of stacks which differ with respect to the class of the objects which may be held in the stack. A final example is the collection of classes of multiprogramming systems which differ with respect to the maximum number of programs which are simultaneously active.

These examples indicate that the detail that varies among 78

the different classes can concern many different attributes of the classes -- the possible "values" of objects, the "types" of the subcomponents or the number of subobjects. The examples also indicate that what varies is a relatively minute detail about the classes -- if the differences were major, then the class variants would be made distinct.

To allow succinct definition of a set of classes which are basically the same but differ with respect to some relatively minor detail, DDN permits parameterized class definitions. The parameters are called qualifiers and are identifiers which may be used at any point within the class definition where a user-defined symbol may appear. When an object of the class is instantiated, values are associated with each of the qualifiers and these values are used in instantiating the object.

For example, suppose that each [knowledge_source] object is activated by one of a number of signals sent from the data base. Each signal indicates that the value of some data base entry of interest to the [knowledge_source] object has changed. Suppose further that each [knowledge_source] object has a number of subobjects which receive the signals and can simultaneously perform the function of the (sub)system -- this allows the signals to be serviced at a faster rate. The variability with respect to the number of data base entries that are of interest and the number of parallel subobjects which can service the signals may be specified in DDN as follows:

```
QUALIFIERS;
  #_values, /* the number of entries of interest */
  #_servers /* the number of parallel servicers */
  END QUALIFIERS;
```

Declaring an object to be of class [knowledge_source(3,5)] would associate the value 3 with the qualifier #_values and the value 5 with the qualifier #_servers. This would lead (with appropriate use of the qualifiers inside the class definition) to the creation of an object which could be signalled because of the changing of any one of three data base entries and which could service at most five signals at a time.

Subcomponents

The subcomponents of each object in a class are declared within the subcomponent textual unit for the class. The declarations given in the subcomponent textual unit are similar to declarations which would appear in a program in some programming language -- they specify names for the objects and indicate the types of the objects, thereby implicitly giving the attributes of the objects. But the declarations in a subcomponent textual unit are always interpreted at instantiation time, prior to "execution" of the system, whereas in many programming languages declarations may give templates for the dynamic, run-time creation of objects. Programming languages generally have a rich set of pre-defined data types (or classes, to use the terminology of DDN). In DDN, however, there are only a few predefined classes (which are discussed in [6] and [7]) and objects are usually declared to be of some user-defined class. Also, programming languages generally

provide a large number of data structuring constructs whereas DDN provides only multi-dimensicnal arrays. All of these differences stem from the fact that DDN is for the description of designs rather than programs -- designs are abstractions of a system and thus the classes are most appropriately developed by the designer rather than predefined within the description language.

The following is a possible subcomponent textual unit for the class [knowledge_source]:

```
SUBCOMPCNENTS;
  activator OF [pre_condition (#_servers) ],
  manipulator ARRAY [ 1::#_servers ] OF [data_base_modifier],
  coordinator OF [semaphore ]
  END SUBCOMPONENTS;
```

The subcomponent of major importance is the array of [data_base_modifier] objects. These objects are subsystem objects which inspect and modify information in the data base. The declaration of the subscript range for the manipulator subcomponent refers to a predefined monitor class, called interval, and indicates that subscripts may lie in the range 1 to #_servers. (Note that #_servers, being one of the qualifiers, will receive a value whenever an object of the class [knowledge_source] is instantiated.)

The interactions among these subcomponents do not concern us here since we are focussing upon the description of system organization rather than system operation. However, a short explanation of their intended interaction is needed to help in understanding the example. The objects in the subcomponent

manipulator are activated by the activator subcomponent. This subcomponent receives the signals coming from the data base and determines which of the manipulator objects should be activated. The coordinator subcomponent is an object of the monitor class [semaphore] which the manipulator objects use to coordinate their concurrent accesses to the data base. As its class name indicates, it functions as a semaphore and is used by the objects in the manipulator subcomponent to mutually exclude their interactions with the data base.

Scope of Identifiers

Subcomponent names are local to the definition of the class in which they are declared. Thus, references to the subcomponents may appear in almost any other textual unit in the class definition (exceptions are noted in [6] and [7]). While the scope of subcomponent names normally should be purely local to the class definition, it is sometimes required that the subcomponents be able to be referenced outside the class definition. Subcomponents may therefore be declared to be "visible" which means that they can be referenced at any point at which an object of that class is declared. For instance, if the coordinator subcomponent had been declared as:

coordinator OF [semaphore] VISIBLE

then this subcomponent could be referenced wherever an object of class [knowledge_source] has been declared. If, in some other class, a [knowledge_source] object were declared with the name source1, then the coordinator subcomponent of source1 could be

referenced as:

source1 | coordinator

The repeated use of the selection operator "|" specifies a selection down through many levels of subcomponentry. The only restriction is that at each step of selection, the selected subcomponent must be visible.

Local Subcomponents

Names of subcomponents are local to the class definition, but are global to the textual units nested within the definition. DDN also allows the definition of subcomponents which are local to one of these textual units. For example, part of the definition of a monitor class is the specification of the procedures which may be invoked upon the objects in the class. It is quite natural to declare a collection of subcomponents which are local to a procedure.

Local subcomponent declarations have the same form as subcomponent declarations. Local subcomponents may not, however, be declared to be visible since this would be contradictory to their being private to a part of a class description.

Component Initialization

In programming languages, variable initialization may be automatically done or the rules of the language may specify that all variables initially have a special value "undefined". In DDN, initial values for objects may not be specified and the designer using DDN must specify the algorithms for the initialization of objects. DDN provides some help in specifying the initialization of monitor objects. Part of a monitor class definition may be designated as an initialization procedure which would be automatically invoked at the beginning of system execution.

Instantiation Control

Using subcomponent textual units to guide the instantiation process gives rise to an instantiation tree since subcomponent textual units do not permit the description of subcomponent sharing. Sharing arises when two objects are described, for convenience or clarity, as if they were distinct, but they are really the same object during the running of the system. Another situation in which sharing arises is when all subobjects y which are a part of objects in some class [x] are to be the same object during the running of the system.

In the example, each [knowledge_source] object has a coordinator subcomponent which the [data_base_modifier] objects use as a mutual exclusion semaphore to coordinate their

modifications of the infcrmation in the data base. This object is really a subcomponent of each of the manipulator objects. It need not be described as a subcomponent of [knowledge_source] objects; this highlights that it is a single object shared among all of the manipulator objects.

One way to describe this sharing of the coordinator subcomponent is to first describe [data_base_modifier] objects as each having a [semaphore] subcomponent:

```
[data_base_modifier]: SUBSYSTEM CLASS;
  SUBCOMPONENTS;
    mutex OF [semaphore]
    END SUBCOMPONENTS;
  END CLASS;
```

This is a redundant declaration since the object is already declared within the [knowledge_source] class definition. But its declaration here is an explicit statement that a [semaphore] object is needed. Also, with this definition, the description of [data_base_modifier] objects may explicitly indicate the manner in which they utilize objects which are actually in the environment in which they run.

To complete the description of the sharing of the coordinator subcomponent, an instantiation control textual unit may be nested within the definition of the class [knowledge_scurce]:

```
INSTANTIATION CONTROL;
  FOR ALL i IN [1::#_servers]
    SAME (coordinator, manipulator(i)|mutex);
    END FOR;
  END INSTANTIATION CONTROL;
```

This textual unit specifies that the mutex sub-subcomponents and

the coordinator subcomponent are all the same object. Thus, for each object of class [knowledge_source] there is only one object of class [semaphore] and any reference to either the subcomponent coordinator or the mutex sub-subcomponents is a reference to this single, shared object. An instantiation graph that would be created by this instantiation control textual unit is given in figure 1. The multiple labels on the node for the

---

k_s OF [knowledge_source(3,2)]

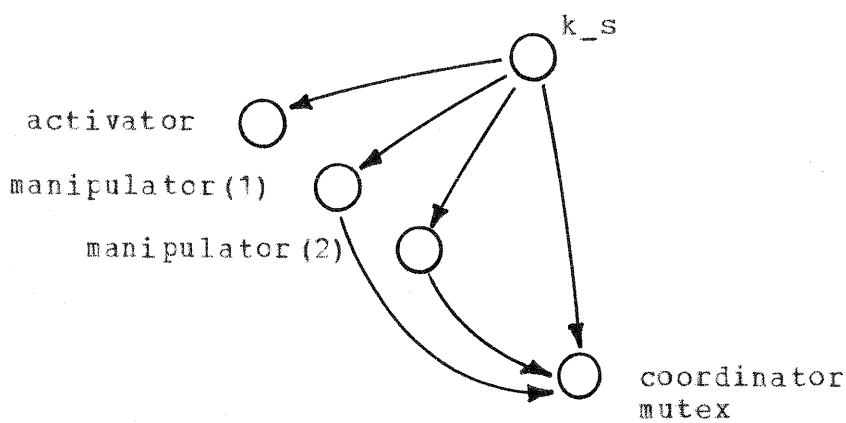

Figure 1

[semaphore] object indicate that it may be referenced by different names from within different objects.

Note that the mutex subcomponents need not be declared as visible. Through the instantiation control textual units, the designer is controlling the configuration of the system. In doing this, he is given the ability to select any object which is a part of the objects in the class being defined.

Thus, a designer has broader powers in configuring the system than he has in preparing the code that executes during system operation. Of course, the designer may misuse this additional power and introduce errors. Rules could be specified to reduce the possibility of errors -- for example, it could be specified that if an accessible object is equated to an inaccessible one, then the accessibility to the first object is lost. But, such rules have not been included in DDN since we have attempted to minimally constrain the design methods a designer may employ. Various methodological constraints may be enforced upon users of the DREAM system [9], but these constraints are not levied by their definition in the DDN language.

An alternative specification would be:

```
INSTANTIATION CONTROL;
  SAME (coordinator, [data_base_modifier]|mutex);
  END INSTANTIATION CONTROL;
```

since reference to a class name may be made in a selector used in an instantiation control textual unit. Only objects which are a part of objects in the class being defined may be selected in this manner. Thus the reference in the last example does not refer to <u>all</u> objects of class [data_base_modifier] but rather <u>only</u> those declared within [knowledge_source]. Another alternative equivalent to those already given is:

```
INSTANTIATION CONTROL;
  SAME ([semaphore],
        [data_base_modifier]|[semaphore]);
  END INSTANTIATION;
```

This could have a different effect, however. If there were another [semaphore] subcomponent of [data_base_modifier] 87

objects, then it would also be selected in addition to the mutex subcomponent.

While the instantiation control textual units given above may sometimes lead to correctly operating [Knowledge_source] objects, proper synchronization generally requires that all mutex sub-subcomponents in all objects of class [knowledge_scurce] be the same object. Achieving this would require a SAME statement which selects all mutex sub-subcomponents in all [knowledge_source] objects. This SAME statement would be very complicated and a more straightforward solution is provided by using another instantiation command:

```
INSTANTIATION CONTROL;
  UNIQUE (coordinator);
  SAME (coordinator, [data_base_modifier]|mutex);
  END INSTANTIATICN CONTROL;
```

This gives the correct structure since the UNIQUE statement leads to there being only one coordinator subcomponent which is shared among all objects of the class [knowledge_source]. A UNIQUE statement may refer tc any object which is part of objects in the class being defined and indicates that the selected object is unique cver all objects in the class being defined. An instantiaticn graph which would be created through the use of this instantiation control textual unit is given in figure 2. Notice that a component is unique over instances that were generated by different qualifier values.

k_s_1 OF [knowledge_source(3,2)]
k_s_2 OF [knowledge_source(3,3)]

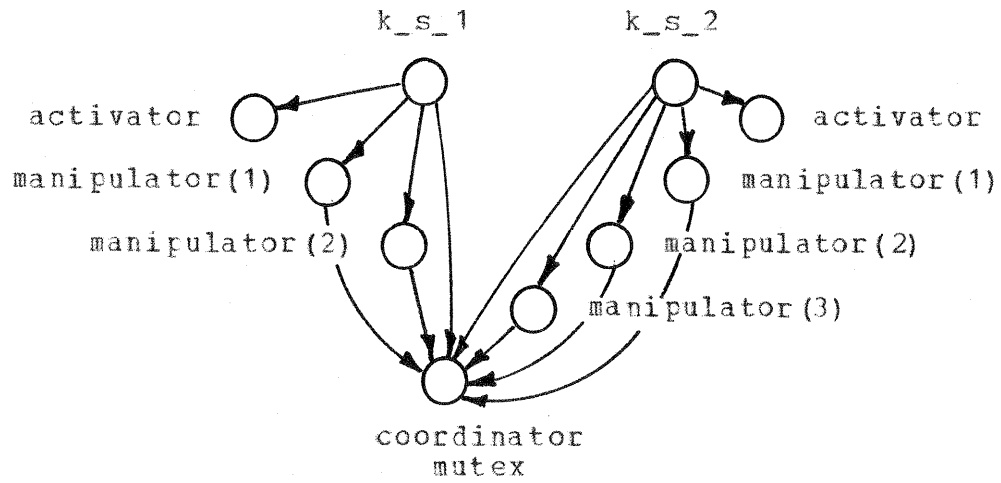

Figure 2

Relationship to Other Approaches to System Description

DDN has been oriented toward the description of a system's design rather toward the programming of a system. This distinction and its ramifications are discussed in general in [11]. In this section, we discuss the effect of this orientation upon the organization description facilities within DDN.

The essence of the difference between design description languages and programming languages is a difference in orientation toward representing a system's behavioral aspects and representing a system's operational detail. In programming, the aim is to prepare a detailed specification of the system's 89

operation and the system's behavioral characteristics are only implicitly described. During design, however, operational detail is usually abstracted and the aim is to prepare a description in which a system's behavior is either explicitly represented or may more easily be inferred. Those parts of DDN which allow the explicit description of system behavior are discussed in [8]. The effects of DDN's orientation to design description discussed here relate to the abstraction of operational detail.

The major effect of working with abstract descriptions is that one works with larger operational units. For instance, text editing and file maintenance systems, rather than sorting and string scanning algorithms, are the units of interest. Because these higher-level, larger units do not naturally decompose (at a reasonable level of description) into standard subunits, DDN does not provide the full set of pre-defined data types and data structures which are typically found in programming languages. The user of DDN is able to describe any object's decomposition in terms of objects of other classes which he may explicitly describe. Thus DDN bears a close resemblance to programming languages which allow the user to define data types. But DDN allows the user to define abstractions of processing structures as well as abstractions of data structures. Thus the objects declared in a subcomponent textual unit are as likely to correspond to what would be a subroutine in a program as they are to correspond to a data fragment.

Qualifiers in DDN provide a primitive macro-expansion facility. In programming languages, macro-expansion facilities provide the programmer with the ability to define generalized processing structures. Since this is already available to the DDN user through the class definition facility, a general macro facility is not necessary. Qualifiers therefore provide a simple text-replacement macro facility. The sophistication of this facility is greatly restricted because of the data base management facilities provided by DREAM (see [9]). A DREAM user may enter fragments of a system's description into the data base in an arbitrary order. Since the description fragments are syntax checked on entry, it is impossible to allow a qualifier to be used in place of more than one lexicographic unit since the definition of the expansion of the qualifier may not be available. Qualifiers therefore provide a relatively unsophisticated macro facility, but do serve well their intended purpose of allowing the definition of generalized classes.

It is in the control of instantiation that DDN differs most extensively from programming languages. First, as noted previously, DDN does not provide for the dynamic, run-time creation of objects. Second, the static instantiation control provided in programming languages is usually implicit and intended for efficiency of storage utilization (e.g., block-structure and its implications for the reuse of storage).

Instantiation control is provided in DDN to allow the description of object sharing. In programming languages this is

usually provided either by a call-by-reference or call-by-name parameter passing mechanism or by some variant of a capability scheme. But, in DDN object sharing stems less from this need for dynamic sharing and more from a need to equate objects that for ease or clarity of description have been defined as if they were separate entities. In the [knowledge_source] example, the coordinator subcomponent could be passed as an argument in invoking operations upon the [data_base_modifier] objects. But, if this were taken as a literal description of the system, then a straightforward implementation would contain much more overhead than necessary. Also, since DDN allows the explicit description of the processing relationships among subcomponents (see [7]) but only an implicit description of the relationships between subcomponents and objects whose identity becomes known at run-time, the description of the class [data_base_modifier] would be less clear were its mutex subcomponent to be described as a parameter received at run-time. Thus, DDN allows the description of sharing in the dynamic manner provided by parameter passing but also provides the ability to control instantiation directly so that parameter passing need not be misused to describe more static sharing of objects.

Summary

DDN has been developed as the design language in DREAM, a tool to aid designers of large-scale systems, and thus differs from traditional programming languages. DDN shares with programming languages the ability to describe a system's

operation but allows descriptions in which the system's operation is abstractly specified. This orientation to abstract description leads to constructs which force the DDN user to explicitly describe the composition and organization of a system.

DDN provides several constructs for describing a system's organization. Chief among these is a construct for defining the composition of a class of objects in terms of their subcomponents which are objects of other classes. DDN provides a primitive macro facility to allow the definition of generalized classes. DDN also provides for the explicit description of object sharing during the execution of the system.

Acknowledgements

The portions of DDN covered in this report were developed with the help of Dirk Kabcenell and Mark Welter. The development of the instantiation control constructs was indirectly contributed to by Victor Lesser, through his work on a control language for collections of concurrent processes.

References

1. Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction Mechanisms in CLU. Comm. ACM, 20, 8 (August 1977), 564-576.

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

3. Dahl, O., and Nygaard, K. SIMULA - An Algol-based simulation language. Comm. ACM, 9, 9 (September 1966), 671-678.

4. Hoare, C.A.R. Notes on data structuring. In Dahl, Dijkstra and Hoare, Structured Programming, Academic Press, New York, 1973.

5. Riddle, W.E., Wileden, J.C., Sayler, J.H., Segal, A.R., and Stavely, A.M. Behavior modelling during software design. IEEE Trans. on Software Engineering, SE-4, 4 (July 1978), 283-292.

6. Riddle, W.E., Sayler, J.H., Segal, A.R., Stavely, A.M. and Wileden, J.C. Abstract monitor types. Proc. Specification of Reliable Software Conf., Boston, April 1979, pp. 37-43.

7. Riddle, W.E., Sayler, J.H., Segal, A.R., Stavely, A.M., and Wileden, J.C. A description scheme to aid the design of collections of concurrent processes. Proc. 1978 National Computer Conf., Anaheim, Calif, June 1978, pp. 549-554.

8. 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.

9. Riddle, W.E., Sayler, J.H., Segal, A.R., and Wileden, J.C. An introduction to the DREAM software design system. Software Engineering Notes, 2, 4 (July 1977), 11-23.

10. Riddle, W.E. DDN User's Guide. RSSM/37, Dept. of Computer Sci., Univ. of Colorado at Boulder, in preparation.

11. Riddle, W.E., Sayler, J.H., Segal, A.R., Stavely, A.M., and Wileden, J.C. DREAM - A software design aid system. In Moneta, J., (ed.), Information Technology, JCIT-3/North-Holland Pub. Co., August 1978.

12. Fennel, R., Lesser, V.R. Parallelism in artificial intelligence problem solving: A case study of HEARSAY II. IEEE Trans. on Computers, C-26, 2 (February 1977).

old

References

1. B. H. Liskov, et al. Abstraction Mechanisms in CLU. Proc. ACM Conf. on Language Design for Reliable Software, Raleigh, N.C., April 1977. (To appear in Comm. ACM.)

2. P. Henderson, et al. The TOPD System. Tech. Report 77, Computing Laboratory, University of Newcastle upon Tyne, England, September 1975.

3. O. Dahl and K. Nygaard. SIMULA -- an ALGOL-Based Simulation Language. Comm. ACM, 9, 9 (September 1966), 671-678.

4. C.A.R. Hoare. Notes on Data Structuring. In Dahl, Dijkstra and Hoare, Structured Programming, Academic Press, New York, 1973.

5. W. Riddle, J. Sayler, A. Segal and J. Wileden. An Introduction to the DREAM Software Design System. Software Engineering Notes, 2, 4 (July 1977).

6. W. Riddle. Abstract Monitor Types. RSSM/41, Dept. of Computer Science, Univ. of Colorado at Boulder, in preparation.

7. W. Riddle. Abstract Process Types. RSSM/42, Dept. of Computer Science, Univ. of Colorado at Boulder, November, 1977.

8. J. Wileden. Behavior Specification in a Software Design System. RSSM/43, Dept. of Computer and Comm. Sciences, Univ. of Michigan, in preparation.

9. A. Segal. Design Description Management. RSSM/45, Dept. of Computer and Comm. Sciences, Univ. of Michigan, in preparation.

10. W. Riddle. DDN User's Guide. RSSM/37, Dept. of Computer Science, Univ. of Colorado at Boulder, in preparation.

11. J. Sayler. Philosophy of the DREAM System. RSSM/39, Dept. of Computer and Comm. Sciences, Univ. of Michigan, in preparation.

12. R. Fennel and V. Lesser. Parallelism in Artificial Intelligence Problem Solving: A Case Study of Hearsay II. IEEE Trans. on Computers, C-26, 2 (February 1977), 98-111.