NPS52-85-005

NAVAL POSTGRADUATE SCHOOL

Monterey, California

A SIMPLE SOFTWARE ENVIRONMENT BASED ON OBJECTS AND RELATIONS

Bruce J. MacLennan

April 1985

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R.H. Shumaker
Superintendent

D. A. Schrady
Provost

The work reported herein was supported by Contract from the Office of Naval Research.

This report was prepared by:

BRUCE J. MACLENNAN
Acting Chairman
Department of Computer Science

KNEALE T. MARSHALL
Dean of Information and
Policy Science

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-85-005
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): A SIMPLE SOFTWARE ENVIRONMENT BASED ON OBJECTS AND RELATIONS
5. TYPE OF REPORT & PERIOD COVERED
6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): Bruce J. MacLennan
8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61153N; RR014-08-01; N0001485WR24057

11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Arlington, VA 22217
12. REPORT DATE: April 1985
13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
15. SECURITY CLASS. (of this report)
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Object-oriented programming, programming environments, software engineering environments, production rules, production systems, entity-relationship approach, software prototyping, knowledge representation, logic programming, simulation languages, rule-based systems, knowledge base, fifth generation languages, classification

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper presents a simple programming system based on a clear separation of *value-oriented programming* and *object-oriented programming* [MacLennan82, MacLennan83a]. The value-oriented component is a conventional functional programming language. The object-oriented component is based on a model of objects and values connected by relations, and on production system-like rules that determine the alteration of these relations through time. It is shown that these few basic ideas permit simple prototyping of a software development environment.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A SIMPLE SOFTWARE ENVIRONMENT BASED ON OBJECTS AND RELATIONS

Bruce J. MacLennan

Computer Science Department

Naval Postgraduate School

Monterey, CA 93943

1. Abstract

This paper presents a simple programming system based on a clear separation of *value-oriented programming* and *object-oriented programming* [MacLennan82, MacLennan83a]. The value-oriented component is a conventional functional programming language. The object-oriented component is based on a model of objects and values connected by relations, and on production system-like rules that determine the alteration of these relations through time. It is shown that these few basic ideas permit simple prototyping of a software development environment.

2. Introduction

2.1 Goals

Our goal has been to design a programming system that combines in one system the best features of value-oriented programming and object-oriented programming. The following two sections discuss these two programming paradigms and their advantages and disadvantages. This is followed by a description of a paradigm that attempts to combine the best features of both. A programming system based on and supporting this paradigm is then discussed. The system is illustrated by several applications to components of programming environments.

2.2 Value-Oriented Programming

There are a number of possible definitions of *value-oriented programming* [MacLennan82, McGraw82, MacLennan83a]. It is most accurately considered programming without the use of explicit state changes. Since the assignment operation is the major programming language construct for effecting state changes, value-oriented programming has also been characterized as *assignment-less programming*.

Also, since variables are the things that assignment operations change, and are the media in which the state is represented, value-oriented programming has also been characterized as *variable-less programming*. There are a number of species of value-oriented programming, including applicative programming [Landin64, Landin66, Burge75], functional programming [Backus78, Henderson80, MacLennan86], equational programming [Hoffman84], logic programming [Kowalski79] and relational programming [MacLennan81, MacLennan83b].

Variables and the assignments to them are some of the most common elements of programs in conventional languages. Why would one want to eliminate them? The advantages of value-oriented programming include ease of program derivation, ability for algebraic manipulation, ease of proof of properties, ease of analysis, evaluation order independence, and very-high-level operations. Examples of value-oriented programming languages include "pure LISP," PROLOG, KRC and Backus's FP.

Value-oriented programming is not without disadvantages. These result directly from the absence of state changes, since this means it is hard to deal with temporal change. Thus, it is difficult to program those applications for which temporal change is an essential element. Examples of these applications include databases, graphics, realtime programs, interactive systems and operating systems. Many of the components of programming environments fall into the above categories, such as editors, version control systems, debuggers, documentation management systems, configuration management systems, etc.

2.3 Object-Oriented Programming

There have been many definitions of *object-oriented programming* [Lomet76, Lomet80, Robson81, Rentsch82, MacLennan82, MacLennan83a]. One view, perhaps the most common, is that the essence of object-oriented programming is data abstraction and information hiding. This seems to be the view, for example, in [Buzzard85]. We prefer to call such languages *data abstraction languages*.

This paper takes a different view of object-oriented programming, a view proposed by Alan Kay [Kay77] and others, namely, that the essence of object-oriented programming is *simulation*. This view is particularly appropriate to systems that must deal explicitly with the passage of time and the alteration of objects through time. These systems include interactive systems, graphics systems, operating systems, file systems, editors, database systems and version control systems. Indeed, all the major

components of a software development environment fit nicely into the simulation paradigm and the object-oriented view [Robson81].

Unfortunately, conventional object-oriented languages, such as Simula [Dahl70] and Smalltalk [Goldberg83], are built upon the framework of a conventional procedural language, which does not take full advantage of the simulation paradigm.

The approach to object-oriented programming taken in this work directly supports the simulation paradigm. This can be summarized as follows:

- Computer objects correspond to external objects.
- The behavior of computer objects models the behavior of external objects.
- Objects are classified according to their behavior.

The advantages of object-oriented programming languages result from their being based on the simulation paradigm. In particular, they are well-suited to deal with time and changes of state in time. Thus they are ideal for dealing with applications for which temporal change is an essential element, such as programming environments, databases, graphics systems, etc. [Robson81].

Object-oriented programming also seems to be quite natural, especially for those without a lot of computer experience [Goldberg77]. This is probably because there is a close correspondence between thinking about computer objects and real-world objects. In many situations the object-oriented program is derivable from real-world situation it is intended to model.

The disadvantages of object-oriented programming are the advantages of value-oriented programming: (1) it is difficult to reason about things that change in time; (2) object-oriented languages provide little ability for algebraic manipulation (although this perhaps could be remedied); and (3) the analysis of object-oriented programs can be hard.

2.4 A Solution: Dyad

Based on these considerations we are developing a programming paradigm, and a programming system to support it, that combines the advantages of object-oriented programming and value-oriented programming. It is called *Dyad* to reflect the fact that it is a whole comprising two distinct units:

- A (alpha) – An *applicative* (value-oriented) component
- Ω (omega) – An *object-oriented* component

Thus the Dyad system accommodates both value-oriented programming and object-oriented programming. The goal of this research is to investigate the advantages and disadvantages of programming in a language, such as Dyad, that clearly separates yet supports both the value- and object-oriented paradigms.

We will not describe here the applicative language (A), since it is conventional and similar to previous applicative languages (KRC, SASL, etc.). The object-oriented language (Ω) is unconventional, however, and will be described in some depth in the following section.

3. The Object-Oriented Component: Ω

3.1 Structure of Ω

In designing the object-oriented component of Dyad, we have attempted to deal directly with the needs of simulation, and not to include any feature just because it is found in other languages. There are several results of this fundamental approach. First, Ω lacks many of the accouterments of conventional languages, such as variables, assignment statements and most control structures. Second, concurrency is the norm; as in the real world being simulated, things happen sequentially only if specifically arranged to do so.

We turn now to the structure of Ω. There are two major issues: The nature of the simulated universe of objects and their relationships, and the means for describing how this universe evolves in time; i.e., for describing the behavior of the objects in terms of the relationships. These issues are discussed in the following two sections.

3.2 Object-Relationship Space

It will be seen that the object-relationship space is similar to several familiar ideas including the knowledge representation languages (semantic nets) used in expert systems [Findler79], the entity-relationship approach to databases [Chen76], and relational databases [Codd70]. Thinking of these will simplify understanding the Ω space.

The Ω object-relationship space is populated by a number of atomic objects connected by relationships. The only properties an object has are those it gets by virtue of the relationships: this is what we mean by the objects being *atomic*. Besides objects being related to objects, objects can also be related to values by the relationships; this provides the connection between the object-oriented and value-oriented components of Dyad.

Relationships are usually expressed in a predicate-logic style. For example, if M and T are particular objects, then a part of the state of a simulation might be expressed by the following relationships:

Missile (M), Moving (M), Position ([25,60], M),

Target (T), ¬Destroyed (T), Position ([18,32], T)

This can be read, "M is a missile, M is moving, [25,60] is the position of M, T is a target, T is not destroyed, and [18,32] is the position of T."

3.3 Transaction Rules

We have discussed how the state of the simulated universe is expressed as atomic objects connected by relationships. We now turn to the transaction rules, which are used to describe state changes. These rules are similar to the production rules used in many AI applications [Newell72]. They are also similar to logic programming clauses [Kowalski79] (although there is no backtrack) and, most relevantly, to causal laws.

The relation of the transaction rules to causal laws can be understood by considering the description of causal relationships in the real world. There are two kinds of causal laws that can be used for this purpose: *entity-oriented* laws and *event-oriented* laws.

In the entity-oriented (or object-oriented) approach the law describes how an object of a given kind behaves in situations of a given kind. Hence, causal laws of this sort are associated with classes of objects. Languages such as Smalltalk and Simula are based on this model of causality.

An alternate model of causality, the event-oriented, says that a causal law relates two kinds of events. Causal laws do this by stating the manner in which events of the first kind cause events of the second kind. This is the model we have adopted in Ω. The event-oriented approach is more general

than the entity-oriented approach, and in fact subsumes it. The event-oriented approach also permits the convenient description of causal relationships in which two or more objects participate as equal partners. Such relationships do not have to be considered properties of any of the objects. This is a problem in Smalltalk, Simula and similar languages [Goldstein80].

Thus our goal is to describe causal relationships in terms of two classes of events standing in the relationship of cause to effect. That is, a causal law tells us how a *present* situation (set of relationships) leads to a *future* situation (set of relationships). Thus a causal law has two parts:

- A *cause,* which describes a present situation in terms of a set of relationships;
- An *effect,* which describes a future situation in terms of a set of relationships.

The *cause part* is just a set of inquiries about tuples in the relationships in the database. For example, the following cause part tests for a particular situation in the state space:

$$\text{Missile}(m),\ \text{Moving}(m),\ \text{Position}(p,m),\ \text{Target}(t),\ \neg\text{Destroyed}(t),\ \text{Position}(p,t) \Rightarrow \quad \cdots$$

This can be read, *"If* there is a missile m, and m is moving, and some p is the position of m, and there is a target t, and t is not destroyed, and p is the position of t, *then* ..."

The *effect part* is a set of transactions that add tuples to, or delete tuples from, relationships. For example, the effect part:

$$\cdots \Rightarrow \neg\text{Moving}(m),\ \text{Destroyed}(m),\ \text{Destroyed}(t).$$

says that the 'Moving' property is to be removed from m, and that the 'Destroyed' property is to be added to both m and t.

3.4 Detailed Discussion

Having described the overall structure of Ω, we now turn to a more systematic presentation of its facilities. The presentation here will be informal. The formal syntax will be found in Appendix A; the formal (denotational) semantics is presented in [MacLennan83a].

The basic construct is the *rule,* which has the form

$$\textit{cause} \Rightarrow \textit{effect}$$

The *cause* describes a possible situation in a space of objects connected by relations. If that situation holds, then the rule may be applied, which means that the actions described by its *effect* part will be performed. Rules are executed *indivisibly,* which means that it is guaranteed that the situation still holds when the actions are performed.

There is normally no order implied between rules; they can be tested in any order. However, rules can be connected by the word **else** when a particular order must be imposed:

$$\begin{aligned} & cause_1 \implies effect_1 \\ \textbf{else}\ & cause_2 \implies effect_2 \\ & \vdots \\ \textbf{else}\ & cause_n \implies effect_n \end{aligned}$$

In this case, the second and succeeding rules are tried only when the preceding rules have failed.

The cause part of a rule describes a situation in terms of one or more *conditions,* which represent the presence or absence of relationships between objects:

$$condition_1,\ condition_2,\ \ldots,\ condition_n$$

All of these conditions must be satisfied before a rule can be applied.

A condition for testing for the *presence* of a tuple in a relationship has the form:

$$primary\ (\ pattern_1,\ pattern_2,\ \ldots,\ pattern_n\ )$$

The *primary* is an expression (usually just an identifier) that evaluates to a relation. The following list of patterns defines a *tuple pattern.* Each pattern in the list can be either a free (i.e., undefined) variable, or an expression containing no free (i.e., only bound) variables. An expression is evaluated during the matching process, and matches a value equal to the result of its evaluation. A free variable will match any value, but becomes bound to that value during the matching process. The special free variable '– ' can be used to match anything without binding a variable name.

A condition for testing for the *absence* of a tuple from a relationship has the form:

$$\neg primary\ (\ pattern_1,\ pattern_2,\ \ldots,\ pattern_n\ )$$

This condition succeeds only if there is *not* a tuple of the specified form in the relation that is the value of the primary.

Finally, as a convenience we permit *cancel* conditions:

$$^{*}primary\ (\ pattern_1,\ pattern_2,\ \ldots,\ pattern_n\ )$$

This tests for the presence of a tuple of the specified form, just as the first kind of condition, but it has a side effect of deleting that tuple if the rule is applied. Although, this could be programmed explicitly, the situation is common enough that it is important to reflect it in the notation.

The effect part of a rule is composed of a sequence of *transactions:*

$$transaction_1,\ transaction_2,\ \ldots,\ transaction_n$$

These transactions can be performed in any order or in parallel. However, their execution forms an indivisible unit.

The transactions are of four kinds: assertions, denials, calls and sequential blocks.

An *assertion* is a transaction of the form:

$$primary\ (\ expression_1,\ expression_2,\ \ldots,\ expression_n\ )$$

Its effect is to add to the relation that is the value of the primary the tuple $< V_1, V_2, \ldots, V_n >$, where each V_i is the value of $expression_i$. Typically these expressions contain variables that were bound in the cause part of the rule.

A *denial* has the form of an assertion preceded by a negation sign:

$$\neg primary\ (\ expression_1,\ expression_2,\ \ldots,\ expression_n\ )$$

Its effect is to *delete* the specified tuple from the specified relation. If this relation does not contain this tuple, then an error condition holds (Ω error handling is beyond the scope of this paper).

A *call* is a transaction of the form:

$$primary\ \{\ expression_1,\ expression_2,\ \ldots,\ expression_n\ \}$$

Its purpose is a form of synchronous communication performed by sending a message through one rela-

tion, and waiting for a reply to be returned through another relation. For example, the call

$$P\{E, F\}$$

has the effect of performing the assertion

$$P(a, E, F)$$

Here a is a newly generated relation that will be used for receiving the reply. This assertion presumably requests some actions to be performed by other rules (which are watching P). When the actions are completed, an acknowledgment or reply will be placed in the a relation, which permits the calling rule to complete. Note that rules containing calls in their effect parts are not considered indivisible. The formal definition of calls through their reduction to basic rules is described in [MacLennan83a].

The last kind of transaction is a sequential block, which has the form:

$$\{\ statement_1;\ statement_2;\ \cdots;\ statement_n\ \}$$

The effect of this construct is to execute the component statements in order. A statement is simply a rule, simple or compound, with the additional characteristic that its cause part (and the $\Longrightarrow$) can be omitted. This reflects the fact that in a sequential block the performance of actions may be conditioned solely on the performance of the preceding statements.

A user normally interacts with an Ω system by typing statements. Thus the form of an Ω terminal session is:

$statement_1$;

$statement_2$;

⋮

$statement_n$;

Many of the statements typed interactively are isolated effects containing a single call. For example, to define 'Contents' to be a new private relation, a user would enter:

Define {Private, "Contents", newrel {} };

Finally, we need a means for manipulating groups of rules as a unit; this is the *rule denotation* and has

the form:

$$<< \ compound-rule_1. \ compound-rule_2. \ \cdots \ compound-rule_n. \ >>$$

Notice that each compound rule is terminated by a period. (A compound rule is simply a rule that may contain elses.)

3.5 Alternate Notations

We also permit predicates to be written in a pseudo-natural notation that comes quite close to the informal meaning of the predicates:

> If a missile is moving, a target is not destroyed and the position of the missile is the position of a target
>
> then the missile is not moving, the missile is destroyed and the target is destroyed.

Notice that the notation permits a limited amount of subordination and anaphoric reference. This pseudo-natural notation is actually very simple to parse; it does not require the use of any sophisticated natural-language-understanding techniques.

We have also investigated two-dimensional, form-based notations, since they can be easily understood by many people. These alternative syntactic representations are described in [MacLennan84]; an implementation of a pseudo-natural notation is described in [Ufford85].

4. Naming and Classification

4.1 Naming

Most programming languages have built-in declarative constructs for binding names to their meanings; there are no such constructs in Ω. Rather, naming is implemented *within* Ω by the use of objects, relations and causal laws. We have taken this approach for two reasons. First, in an interactive system bindings are created and destroyed through time. This temporal change implies that name directories are objects, and hence that Ω is the appropriate vehicle for dealing with them. Second, a loose coupling between the naming strategy and the language permits experimentation with alternative naming strategies, which is appropriate in a prototype system such as Dyad.

The current Dyad naming strategy makes use of objects called *directories* that bind names (strings) to values or objects. This is accomplished through two relationships, 'Directory' and 'Binds'. The meaning of Directory(d) is that d is a directory; the meaning of Binds(d,n,x) is that directory d binds name (string) n to object or value x.

Bindings are established by the procedure call:

$$\text{Define } \{d,\ n,\ x\}$$

In the simplest case this procedure is implemented by the following rule, which makes the appropriate entries in Binds:

$$\text{*Define } (a,\ d,\ n,\ x),\ \text{Directory } (d) \Longrightarrow\ \text{Binds } (d,\ n,\ x),\ a\ (n)$$

(The extra parameter a is used for return from the procedure; see calls in Section 3.4.) In practice it is desirable to allow the *rebinding* of already bound names. This is easily accomplished by the compound rule:

*Define (a, d, n, x), Directory (d), *Binds (d, n, y)

$\Longrightarrow$ Binds (d, n, x), a (n + " redefined")

else *Define (a, d, n, x), Directory (d) $\Longrightarrow$ Binds (d, n, x), a (n + " defined")

The first rule handles the case where n is already bound to some y; this binding is discarded. In either case the caller a is returned a message indicating that n was (re)defined.

When Ω executes a rule (more correctly, when the rule is activated), it must look up the meanings of all the names that occur in that rule. Although this lookup operation is built into the system, its effect is the same as executing the procedure call 'Lookup$\{n,d\}$', where d is the "current" directory. The default implementation of Lookup is:

$$\text{*Lookup } (a,\ n,\ d),\ \text{Directory } (d),\ \text{Binds } (d,\ n,\ x) \Longrightarrow\ a\ (x)$$

That is, we return to the caller a the object x to which the name n is bound by the directory d.

That the Dyad naming system is defined *within* Ω adds considerable flexibility to the system. To implement a different naming strategy all we need do is change the definitions of Define and Lookup.

For example, if we wanted to permit lookup operations to follow a path from directory to directory, we could do this by defining a new relation Path and by making the appropriate change to Lookup:

*Lookup (a, n, d), ¬Binds (d, n, x), Path $(d, e) \Rightarrow$ Lookup (a, n, e)

else *Lookup (a, n, d), Binds $(d, n, x) \Rightarrow a\ (x)$

else *Lookup $(a, n, d) \Rightarrow a$ (Nil).

The first rule detects if the sought name is not bound in the current directory, and forwards the search to the next in the path. We have also added here a rule to detect unbound names.

4.2 Information Hiding

In a simulation it is often necessary to implement an object in terms of lower level objects. To preserve the integrity of the simulation it is important to distinguish those objects and relationships that *are* part of the simulation (i.e., correspond to objects and relationships in the system being simulated), from those objects and relationships that are *not* part of it (i.e., do not correspond). There is also a software engineering reason for making this distinction, namely, facilitation of the modular decomposition of the system.

Although both Simula-67 and Smalltalk permit objects to be used in the implementation of other objects, neither language enforces the boundary between these levels of abstraction. Such enforcement is a natural extension that has been included in more recent languages (including newer versions of Simula). It is, I believe, the reason that object-oriented languages are commonly identified with data abstraction languages and languages supporting information hiding.

It is for these reasons – the support of both simulation integrity and modular decomposition – that we have included in Ω facilities for enforcing levels of abstraction. This enforcement is accomplished by controlling the kinds of access to the relations permitted to various parties. In particular, rather than being bound directly to relations, relation names are bound to *capabilities* [Dennis66] referring to the relations. Each such capability may bear from zero to three *rights*, the permitted rights being *read*, *insert* and *delete*. These rights correspond to the ways in which relations can be used in rules: *read* for use in the cause part, *insert* for use in an assertion in the effect part, and *delete* for use in a denial in the effect

part.

An example will illustrate how capabilities can be used to enforce information hiding and levels of abstraction. Suppose that we intend to implement a system service such as an editor. It is intended that users be able to invoke this service by a procedure call such as 'Exit{X}', where X is the object to be edited. This procedure call is equivalent to the assertion 'Edit(A,X)', where A is a relation-surrogate for the agent performing the call. It can be seen that the call entails placing the tuple $< A,X >$ in the Edit relation, which can be thought of as putting the message $< A,X >$ in the Edit mailbox.

Now, this usage pattern has several consequences. First, users of services must have *insert* rights to the relations used for requesting these services. Second, users of services should not have *delete* rights for these relations, since that would permit them to delete other users' requests (postal patrons can't take mail out of public mail-boxes). Further, users should not have *read* rights, since that would permit them to discover the requests made by other users.

Thus it can be seen that the user of such a service-request relation should have insert-only rights to that relation. On the other hand, the implementor of the service must have full rights to the relation, otherwise it will be unable to read and delete the requests as they are serviced.

Arrangements such as these are easily accomplished by means of the directory structure previously described. For example, suppose that we have two directories: 'Public' contains all the names generally available to users, and 'EditDir' contains all of the (public and private) objects and relations needed to implement the editor. The private version of the request relation is created and named by:

Define {EditDir, ''Requests'', newrel{}}

The system procedure 'newrel' creates a new relation and returns a capability for that relation that bears full rights. Within the implementor's directory (EditDir) this capability will be known as 'Requests'.

To make the request relation available to potential users, we must give it a name in a publicly accessible directory. Since potential users should have insert-only rights to this relation the public name 'Edit' is bound to an insert-only capability for Requests. This is accomplished by executing the follow-

ing call with EditDir as the current directory:

Define {Public, "Edit", insertonly {Requests}}

The system procedure 'insertonly' is used to produce the reduced-rights capability that is made publicly available.

The basic approach illustrated above can be extended to more complicated directory structures and more complicated sharing and access policies.

4.3 Hierarchical and Nonhierarchical Classification

Classification and simulation go hand in hand. It is no coincidence that a class construct appeared in Simula 67, a simulation language, long before it appeared, or its value was recognized, in other languages. The reason is simple: in a simulation of any complexity it is infeasible to describe the behavior of every individual object. Rather, it is necessary to group objects into classes of similarly-behaving individuals, so that their common behavior can be described just once. At root this is the same reason we seek scientific laws.

We have already shown how Ω supports a simple (one level) classification system. A rule such as

$$*\text{Define}\ (a, d, n, x), \text{Directory}\ (d) \implies \text{Binds}\ (d, n, x), a\ (x)$$

is part of the description of the behavior of Directory objects.

The integrity of Directories as abstract objects can be ensured by making the following rights publicly available:

Define – insert-only

Directory – read-only

Binds – no rights

'Define' has insert-only rights because it is used for requesting services; 'Directory' has read-only rights because it is used as the tag for Directory objects; and 'Binds' has no publicly accessible rights because it is part of the internal structure (implementation) of Directories.

In more complex systems it is economical to factor common behavior out of several *classes* of

objects; that is, to treat several classes as species under a common genus. To support this *hierarchical classification* Simula and Smalltalk permit classes to be *subclasses* of other classes. In Ω the same result can be achieved by the Path structure previously described. In effect, directories correspond to the classes whose behavior they encapsulate.

Other researchers have described the limitations of a purely hierarchical classification system [Goldstein80]. In these situations it is convenient for a class to be able to have more than one immediate superclass. Nonhierarchical classification can be handled in Ω by arranging for the Path to thread in the desired order through the directories corresponding to the superclasses.

It is our goal that the flexibility of Dyad's naming and capability system will encourage experimentation with nonhierarchical and other classification policies.

5. Applications

To aid investigation and evaluation of this kind of object-oriented programming we have developed a prototype implementation of a programming environment for a simple applicative language. This implementation is described in detail in [MacLennan85]. The following table shows the number of rules required for programming several major components of the system:

Ω Rules	Component
22	Parallel Interpreter
9	Pretty Printer
16	Code Generator
14	Debugger
24	Language-Specific SDE
37	Universal SDE
122	TOTAL

To convey some idea of the complexity of actual rules we show two rules from this programming environment. The first rule is from a parallel expression evaluator. In the pseudo-natural notation it

is:

If a function is the meaning of the operator of an application, given value-1 is the value of the left argument of the application, and given value-2 is the value of the right argument of the application

then the function of value-1 and value-2 is the value of the application.

In the predicate notation it is:

Appl(E), Op(N,E), Left(X,E), Right(Y,E), Meaning(F,N), *Value(U,X), *Value(V,Y)
$\Rightarrow$ Value ($F[U,V]$, E).

Appendix B includes the complete Ω definition of a combined parallel evaluator/unparser for an arithmetic expression language. These rules are in the dialect of Ω accepted by the McArthur interpreter [McArthur84].

A language-independent syntax-directed editor is used as another component of the example system. For example, the following rule moves the cursor to the leftmost daughter of the current node:

*Command(**in**), *Cursor(n,f), Subnode(n,f,m), Type(m,t), LeftField(t,g)
$\Rightarrow$ Cursor(m,g).

The second example is a rule from a different universal (i.e., table driven) syntax-directed editor:

*Command(**in**), CurrentNode(E), Type(V,E),
AssumedTemplate(T,V), FirstSelector(R,T),
Component(X,R,E)

$\Rightarrow$ SetCurrentNode(X), Command(**display**).

The syntax-directed editor is explained in detail in [MacLennan85]. This report also discusses the application of similar techniques to the interactive creation and modification of other data structures represented by relations.

For another example we consider a simple version of Reed's file system [Reed78]. The following rules allow a value to be read, effective at any time, and record the time of the latest read:

*Read(A,r), Value(r,T,x), LastRead(r,t), $T \leqslant t$ $\Rightarrow$ $A(x)$.

*Read(A,r), Value(r,T,x), *LastRead(r,t), $T > t$ $\Rightarrow$ LastRead(r,T), $A(x)$.

The following rules allow an update of a value effective at a given time, but only if that time is later than the time of the last read:

*Update(r,T,y), LastRead(r,t), $T > t$ $\Rightarrow$ Value(r,T,y).

*Update(r,T,y), LastRead(r,t), $T \leqslant t$ $\Rightarrow$ AbortUpdate(T).

Since the users are considered objects in the system, they can interact with the system by performing searches, inserts and deletes on relations for which they have the appropriate capabilities [Dennis66].

6. Implementation

6.1 Declarations and Second Order Relationships

We are investigating several techniques to keep the implementation of the Dyad system acceptably efficient. These involve both the more efficient representation of relations, and more efficient determination of the rules eligible for application.

Both goals are accomplished by giving users the ability to put declarative information about relationships into the object-relationship space. These *second order relationships* have a role similar to declarations in conventional languages, that is, they can be used for documentation, error checking and improved performance. Unlike declarations in most other languages, they are completely optional. The philosophy is that if programmers have information about the relationships that they think it would be useful to tell the system or other users, then they can do so. However, there is no obligation for them to provide this information or for the system or other users to make use of it.

We give a few examples of these second order relationships. It is frequently useful to know the adicity of a relationship and the types of values and objects it can relate. This information can be made available in the 'Degree' and 'Domain' relationships. For example, 'Degree(R,n)' means that the degree of R is n, that is, all the tuples in R are n-tuples. Similarly, 'Domain(P,k,R)' means that domain k (i.e., the kth argument position) of relationship R is P, where P either is a type

membership predicate (Boolean valued total function), or is degree 1 relationship (finite predicate).

Another useful second order relationship is 'Indexed': 'Indexed(R,k)' means that R is indexed on its kth argument, that is, the values of the kth argument must be unique. Knowing that a relationship is indexed permits the testing of the cause parts of rules to be done much more efficiently. To see an example of an indexed relationship, consider the following four assertions:

Degree (Op, 2)

Domain (string, 1, Op)

Domain (Appl, 2, Op)

Indexed (Op, 2)

These tell us that relationship 'Op' has two columns, the first of which holds strings and the second of which holds members of the relationship called 'Appl'. Furthermore, Op is indexed on its second column. Thus, if n is an object such that Appl(n) is true, then there is at most one string s such that Op(s,n). In effect Op is a finite partial function, or table, mapping Appl objects to strings.

Relationships representing finite partial functions (tables) are so common that a special abbreviation is provided for them. The assertion 'Function(F,D,R)' is equivalent to the four assertions

Degree (F,2)

Domain (R, 1,F)

Domain (D, 2,F)

Indexed (F,2)

Thus, the previous assertion about Op can be expressed 'Function (Op, Appl, string)', which means that Op is a (finite, alterable) function from Appl objects into strings.

It is our intention to improve performance by allowing various representations for relationships (e.g., linked lists, hash tables, relational databases). Second order relationships allow the programmer to give the system suggestions about the representation to use. For example, 'Repr (Op,hash)' suggests that a hash table be used to represent Op.

6.2 Prototype Implementations

Several prototype implementations of Dyad have been completed. The first prototype was implemented in LISP. This prototype supported only one representation of relationships: lists of tuples.

The second prototype was also implemented in LISP, but supports multiple representations of relations to improve performance. This prototype interpreter accepts rules in a tree form that can be generated either by a parser written in LISP, or by a syntax-directed editor. The table-driven syntax-directed editor in this case is especially interesting because it is written as Ω rules operating on a translation grammar expressed in the Ω object-relationship space. Thus we have successfully involuted the system, having Ω programs generate Ω programs. The table-driven syntax-directed editor is described in [MacLennan85].

The third prototype is intended for increased performance. The interpreter is coded in C and provides access to the UNIX shell for various utility and convenience functions. An interpreter for the applicative language (A component) also has been completed. The third prototype, like the second, operates on rules in a tree form that can be generated either by a universal syntax-directed editor or by a conventional parser. In this case, however, the editor has been programmed in Pascal and the parser generated by YACC. A complete description of the third prototype can be found in [McArthur84]; it is the system on which the example in Appendix B runs.

6.3 Status

The two languages (A and Ω) are stable, but subject to revision based on experience. The Dyad system is being evaluated by being used to describe and prototype a variety of applications. Several of these, including preliminary performance measurements, are described in [McArthur84]. This approach is giving us experience in programming realistic problems. We are also experimenting with changes to Ω to increase its knowledge representation capabilities.

Alternate notations and visual representations for rules are being evaluated. For example, we are investigating a two-dimensional form based notation that allows transactions to be visualized as the arrival and distribution of forms. This might be useful in office automation applications.

7. Acknowledgements

Thanks are due to Heinz McArthur for implementing the first and third prototypes, to Dennis Hall for implementing the second prototype, and to George Tilley for implementing the universal syntax-directed editor used with the third prototype. Support for this research was provided by the Office of Naval Research under contracts N00014-84-WR-24087 and N00014-85-WR-24057.

8. References

[Backus78] Backus, John, Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs, *Comm. ACM 21,* 8 (August 1978), pp. 613-641.

[Burge75] Burge, W. H., *Recursive Programming Techniques,* Addison-Wesley, Reading, 1975.

[Buzzard85] Buzzard, G. D., and Mudge, T. N., Object-Based Computing and the Ada Language, *IEEE Computer 18,* 3 (March 1985), pp. 11-19.

[Chen76] Chen, Peter Pin-Shan, The entity-relationship model – toward a unified view of data, *ACM Trans. Database Sys. 1,* 1 (1976), pp 9-36.

[Codd70] Codd, E. F., A Relational Model for Large Shared Data Banks, *CACM 13,* 6 (June 1970), pp 377-387.

[Dahl70] Dahl, O.-J., Myhrhaug, B. and Nygaard, K., *The SIMULA 67 Common Base Definition,* Publication S-22, Norwegian Computing Centre, Oslo, 1970.

[Dennis66] Dennis, Jack B. and Van Horn, Earl C., Programming semantics for multiprogrammed computations, *CACM 9,* 3 (March 1966), pp 143-155.

[Findler79] Findler, Nicholas V., (ed.) *Associative Networks: Representation and Use of Knowledge by Computers,* Academic Press, 1979.

[Goldberg77] Goldberg, Adele and Kay, Alan, Teaching Smalltalk (Methods for Teaching the Programming Language Smalltalk; Smalltalk in the Classroom), XEROX Palo Alto Research Center report SSL 77-2, June 1972.

[Goldberg83] Goldberg, Adele and Robson. David. *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, 1983.

[Goldstein80] Goldstein, I. and Bobrow, D., Extending Object Oriented Programming in Smalltalk, *Conference Record of the 1980 LISP Conference,* The LISP Conference, August 25-27, 1980.

[Henderson80] Henderson, Peter, *Functional Programming: Application and Implementation,* Prentice/Hall International, Englewood Cliffs, 1980.

[Hoffman84] Hoffman, C. M., and O'Donnell, M. J., Implementation of an Interpreter for Abstract Equations, *Conf. Record Eleventh Annual ACM Symp. on Principles of Prog. Lang.,* January 15-18, 1984.

[Kay77] Kay, Alan C., Microelectronics and the personal computer, *Scientific American 237,* 3 (September 1977), pp 230-244.

[Kowalski79] Kowalski, Robert, *Logic for Problem Solving,* Elsevier'North Holland, 1979.

[Landin64] Landin, P. J., The Mechanical Evaluation of Expressions, *Computer J. 6,* 4 (January 1964), pp. 308-320.

[Landin66] Landin, P. J., The Next 700 Programming Languages, *Comm. ACM 9,* 3 (March 1966), pp. 157-166.

[Lomet76] Lomet, David B., Objects and Values: The Basis of a Storage Model for Procedural Languages, *IBM Journ. Res. and Dev. 20,* 2 (March 1976), pp 157-167.

[Lomet80] Lomet, David B., A Data Definition Facility Based on a Value-Oriented Storage Model, *IBM Journ. Res. and Dev. 24,* 6 (November 1980), pp 764-782.

[MacLennan81] MacLennan, B. J., Introduction to Relational Programming, *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture,* ACM Order No. 556810, pp 213-220.

[MacLennan82] MacLennan, B. J., Values and Objects in Programming Languages, *SIGPLAN Notices 17,* 12 (December 1982), pp 70-79.

[MacLennan83a] MacLennan, B. J., A View of Object-Oriented Programming, Naval Postgraduate School Computer Science Department Technical Report NPS52-83-001, February 1983.

[MacLennan83b] MacLennan. B. J., Relational Programming, Naval Postgraduate School Computer Science Department Technical Report NPS52-83-012. September 1983

[MacLennan84] MacLennan, B. J., The Four Forms of Ω : Alternate Syntactic Forms for an Object-Oriented Language, Naval Postgraduate School Computer Science Department Technical Report NPS52-84-026, December 1984.

[MacLennan85] MacLennan, B. J., Experience with Ω : Implementation of a Prototype Programming Environment, Naval Postgraduate School Computer Science Department Technical Report forthcoming.

[MacLennan86] MacLennan, B. J., *Functional Programming Methodology: Theory and Practice* (tentative title), to be published by Addison-Wesley.

[McArthur84] McArthur, Heinz M., *Design and Implementation of an Object-Oriented, Production-Rule Interpreter,* MS Thesis, Naval Postgraduate School Computer Science Department, December 1984.

[McGraw82] McGraw, James R., The VAL Language: Description and Analysis, *ACM Trans. Prog. Lang. and Sys. 4,* 1 (January 1982), pp. 44-82.

[Newell72] Newell, A. and Simon, H. A., *Human Problem Solving,* Prentice Hall, 1972.

[Reed78] Reed, David P., *Naming and Synchronization in a Decentralized Computer System,* PhD Dissertation, 1978, MIT/LCS/TR-205.

[Rentsch82] Rentsch, Tim, Object Oriented Programming, *SIGPLAN Notices 17,* 9 (September 1982), pp 51-57.

[Robson81] Robson, David, Object-oriented software systems, *BYTE 6,* 8 (August 1981), pp 74-86.

[Ufford85] Ufford, Robert P., *A Translation of an Extensible "Natural" Notation Language into an Object Oriented Language (Omega)* (tentative title), MS Thesis, Naval Postgraduate School Computer Science Department, June 1985.

APPENDIX A: SYNTAX OF Ω

Session = *statement−list* .

statement−list = *statement* ; · · ·

$$\textit{statement} = \left\{ \begin{array}{l} \textit{rule}\ \textbf{else}\ \textit{statement} \\ [\,\textit{cause} \Longrightarrow\,]\ \textit{effect} \end{array} \right\}$$

compound−rule = *rule* [**else** *statement*]

rule = *cause* ⟹ *effect*

cause = [**if** *condition* , · · ·

$$\textit{condition} = \left[\begin{array}{l} * \\ \neg \end{array} \right] \textit{inquiry}$$

inquiry = *primary* (*tupl−pattern*)

$$\textit{tupl−pattern} = \left\{ \begin{array}{l} \textit{pattern}\ ,\ \cdots \\ \textit{pattern}\ :\ \textit{pattern} \end{array} \right\}$$

$$\textit{pattern} = \left\{ \begin{array}{l} \textit{free-variable} \\ \textit{expression} \end{array} \right\}$$

$$\textit{free-variable} = \left\{ \begin{array}{l} - \\ \textit{variable} \end{array} \right\}$$

effect = [*transaction* , · · ·]

$$\textit{transaction} = \left\{ \begin{array}{l} \textit{assertion} \\ \textit{denial} \\ \textit{call} \\ \textit{seq−block} \end{array} \right\}$$

assertion = *predication*

denial = ¬ *predication*

predication = *primary* (*arguments*)

call = *primary* { *arguments* }

arguments = [*expression* , · · ·]

seq−block = { *statement−list* }

expression = [*expression* ∨] *conjunction*

conjunction = [*conjunction* ∧] [¬] *relation*

$$relation = [simplex\ relator]\ simplex$$

$$relator = \{= | \neq | < | > | \leq | \geq \}$$

$$simplex = [simplex\ \{+|-\}]\ term$$

$$term = [term\ \{*|/|\%\}]\ factor$$

$$factor = \left[\begin{array}{c} + \\ - \end{array}\right]\ primary$$

$$primary = primitive\ [: primary]$$

$$primitive = \left\{\begin{array}{l} constant \\ [@\ ]\ variable \\ primitive\ [\ arguments\ ] \\ (\ expression\ ) \\ [\left\{\begin{array}{l} expression\ ,\ \cdots \\ expression : expression \end{array}\right\}] \\ call \\ rule-denotation \end{array}\right\}$$

$$constant = \left\{\begin{array}{l} digit^{+} \\ "\left\{\begin{array}{l} char \\ "" \end{array}\right\}^{*}" \\ \textbf{nil} \end{array}\right\}$$

$$rule-denotation = <<\ \{\ compound-rule\ .\ \}^{*}\ >>$$

APPENDIX B: INTERPRETER/UNPARSER FOR SMALL LANGUAGE

```
!     Rules and associated definitions for a universal

!     interpreter/unparser for a small language of

!     arithmetic expressions.


!     Relations

define {root, "Eval", newrel{}};

define {root, "Check", newrel{}};

define {root, "Value", newrel{}};


define {root, "Appl", newrel{}};

define {root, "Op", newrel{}};

define {root, "Left", newrel{}};

define {root, "Right", newrel{}};

define {root, "Con", newrel{}};

define {root, "Litval", newrel{}};


define {root, "Meaning", newrel{}};

define {root, "Template", newrel{}};

define {root, "Explanation", newrel{}};


define {root, "CurrentNode", newrel{}};


!     Functions

fn Id [x]: x;

fn Sum [x,y]: x + y;

fn Dif [x,y]: x - y;

fn Product [x,y]: x * y;
```

```
fn Quotient [x,y]:
   if y = 0 -> ["error", 1]
   else x / y;

fn IsErrorcode [w]:
   if ~IsList[w] | w = Nil -> Nil
   else first[w] = "error";

fn upSum [x,y]: "(" + x + " + " + y + ")";
fn upDif [x,y]: "(" + x + " - " + y + ")";
fn upProd [x,y]: "(" + x + " x " + y + ")";
fn upQuot [x,y]: "(" + x + " / " + y + ")".

!     Built-in Tables

Meaning (Sum, "+ ");
Meaning (Dif, "-");
Meaning (Product, "x");
Meaning (Quotient, "/");
Meaning (Id, "lit");

Template (upSum, "+ ");
Template (upDif, "-");
Template (upProd, "x");
Template (upQuot, "/");
Template (int_str, "lit");

Explanation ("incomplete program", ["error", 0]);
Explanation ("division by zero", ["error", 1]);
```

```
!       the Rules

define {root, "EvalUnparseRules", < <

        ! Constant node

if *Eval (class, e), Con (e), Litval (v,e), class (f, "lit")
-> Value (class, f[v], e);

        ! Application node:  Analysis rule

if *Eval (class, e), Appl (e), Left (x,e), Right (y,e)
-> Eval (class, x), Eval (class, y);

        ! Application node:  Synthesis rules

if *Value (class, u,x), *Value (class, v,y),
 Appl (e), Op (n,e), Left (x,e), Right (y,e), class (f, n)
-> Check (class, f[u,v], e);

if *Check (class, w, e), ~IsErrorcode [w]
-> Value (class, w, e);

if *Check (class, w, e), IsErrorcode [w],
 Explanation (s, w), *CurrentNode (q)
-> displayn {s}, CurrentNode (e)

> > }.

!       activate the rules

act {EvalUnparseRules}.
CurrentNode (Nil).
displayn {"Evaluator/Unparser loaded"}.
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 — 2

Dudley Knox Library
Code 0142
Naval Postgraduate School
Monterey, CA 93943 — 2

Office of Research Administration
Code 012A
Naval Postgraduate School
Monterey, CA 93943 — 1

Chairman, Code 52
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 — 40

Associate Professor Bruce J. MacLennan
Code 52ML
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 — 12

Dr. Robert Grafton
Code 433
Office of Naval Research
800 N. Quincy
Arlington, VA 22217-5000 — 1

Dr. David Mizell
Office of Naval Research
1030 East Green Street
Pasadena, CA 91106 — 1

Professor Jack M. Wozencraft, 62Wz
Department of Electrical and Comp. Engr.
Naval Postgraduate School
Monterey, CA 93943 — 1

Professor Rudolf Bayer
Institüt für Informatik
Technische Universität
Postfach 202420
D-8000 Munchen 2
West Germany — 1

Dr. Robert M. Balzer
USC Information Sciences Inst.
4676 Admiralty Way
Suite 10001
Marina del Rey, CA 90291 — 1

Mr. Ronald E. Joy
Honeywell. Inc.
Computer Sciences Center
10701 Lyndale Avenue South
Bloomington, MI 55402 1

Mr. Ron Laborde
INMOS
Whitefriars
Lewins Mead
Bristol
Great Britain 1

Mr. Lynwood Sutton
Code 424, Building 600
Naval Ocean Systems Center
San Diego, CA 92152 1

Mr. Jeffrey Dean
Advanced Information and Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040 1

Mr. Jack Fried
Mail Station D01/31T
Grumman Aerospace Corporation
Bethpage, NY 11714 1

Mr. Dennis Hall
New York Videotext
104 Fifth Avenue, Second Floor
New York, NY 10011 1

Professor S. Ceri
Laboratorio di Calcolatori
Departimento di Elettronica
Politecnico di Milano
20133 - Milano
Italy 1

Mr. A. Dain Samples
Computer Science Division - EECS
University of California at Berkeley
Berkeley, CA 94720 1