UMIST OBJ: a Language for Executable Program Specifications

R. M. GALLIMORE[1], D. COLEMAN[1] AND V. STAVRIDOU[2]*

[1] *HP Labs, Hewlett-Packard, Filton Road, Stoke Gifford, Bristol BS12 6QZ*

[2] *Computation Department, UMIST, PO Box 88, Sackville Street, Manchester M60 1QD*

This paper defines the algebraic specification language implemented by the UMIST OBJ system. It also illustrates the use of the language for the definition of abstract, executable specifications of the behaviour of computer programs. The system implements an executable subset of J. A. Goguen's OBJ language, which is based on the algebraic definition of abstract data types. The language permits data types and operations to be defined abstractly, i.e. independently of any particular representation. Moreover, the definitions of an OBJ specification can be treated as an abstract program, by regarding the equations contained in a specification as a set of left-right rewrite rules which may be used to simplify terms. This makes the language useful for formulating and exploring the consequences of abstract designs, and developing relevant parts of the theory of the associated problem domain. The ability to exercise descriptions of the theory of a problem domain is a powerful tool for the programmer. By providing timely feedback on the correctness of design decisions, such use of the language encourages and reinforces the exploration of design possibilities. With these features, UMIST OBJ embodies the foundations of a framework for effective software engineering. It provides an accessible basis for both mechanisable formal notations for program description and semantically motivated support tools.

Received August 1987, revised October 1987

1. INTRODUCTION

The UMIST OBJ system implements an executable subset of the language OBJ developed by J. A. Goguen,[8,11] over a number of years at UCLA and SRI International, California. OBJ as described in Refs 8 and 11 is not yet fully implemented, nor is it available for distribution. Therefore the UMIST OBJ effort has concentrated on developing a widely available and portable version of the language, rather than adding functionality to the original OBJ system, the aim being to allow early and wide experimentation of the language in software engineering projects. Accordingly, the system has been widely distributed (over 60 sites) both in industry and academia, where it has been used with considerable success. Some results are reported later in this paper.

The principal components of an OBJ specification correspond to equationally specified abstract data types, or to applicative definitions of operators which compute over existing data types. We shall proceed by presenting the syntax of the main features of the UMIST OBJ language, to which we shall attach an informal description of its intended semantics. We restrict the syntax of operator application to the parenthesised prefix form of standard functional notation. The syntax of UMIST OBJ is presented in Extended Backus Naur Form (EBNF) notation. The language description is followed by a brief discussion of the (mathematical, proof-theoretic and operational) semantics of UMIST OBJ specifications. We then illustrate a possible role for OBJ specifications in a more general software engineering framework, via a simple, topical example. We also discuss further non-trivial uses of the language in a miscellany of projects. Finally, we describe the implementation details relevant to the versions of the system currently available and report on further work and developments concerning the system.

* To whom correspondence should be addressed at: Department of Computer Science, RHBNC, University of London, Egham Hill, Egham, Surrey TW20 0EX, UK.

2. AN INTRODUCTION TO THE LANGUAGE

Objects

An OBJ specification exhibits a modular structure, whose basic components are called objects. Objects are used for the definition of abstract data types (that is, sets of values together with operators which manipulate those values), and for the abstract description of algorithms.

An OBJECT is introduced by the keyword OBJ followed by an identifier which serves to name the textual unit denoting the OBJECT.

```
⟨object⟩ :: = OBJ ⟨object-id⟩ ['/'⟨used-objects⟩]
              [SORTS ⟨sort-list⟩]
              [OPS ⟨op-declaration-list⟩]
              [VARS ⟨variable-declaration-list⟩]
              [EQNS ⟨equation-list⟩]
          JBO
```

The ⟨*used-objects*⟩ component of an object heading is a list of names of objects which contain the declaration of sorts and/or operators which are referenced within the current object. The end of an object is marked by the keyword JBO.

Sorts

An object may introduce a number of identifiers as names of sorts. Each sort name denotes a set of values called the carrier of that sort. The idea of a sort corresponds roughly to the notion of a type in a programming language such as Pascal.

```
⟨sort-list⟩ :: = ⟨sort-name⟩ {⟨sort-name⟩}
⟨sort-name⟩ :: = ⟨id⟩
```

Sort names are globally available within the object that introduces them and within any objects which subsequently gain access to that object, via the transitive closure of the use relation.

Operator declarations

An object may introduce operators by giving for each operator its name and functionality. Each operator is strongly typed by defining its arity (the number and sorts of its arguments) and (target) sort.

```
⟨op-declaration-list⟩ :: = ⟨op-declaration⟩
                          {⟨op-declaration⟩}
⟨op-declaration⟩ :: = ⟨form-list⟩ ':' [⟨arity⟩] → ⟨sort⟩
⟨form-list⟩ :: = ⟨op⟩ {':' ⟨op⟩}
⟨op⟩ :: = ⟨id⟩
⟨arity⟩ :: = ⟨sort-list⟩
```

An operator of null arity denotes a constant of the target sort of the operator. The following object introduces a primitive sort item, together with nine operators, *i*1 .. i9, which name constants of sort item.

```
OBJ item
SORTS item
OPS
  i1,i2,i3,i4,i5,i6,i7,i8,i9 : → item
JBO
```

Note that the definition of item does not imply any ordering on its elements, only that the elements exist and are distinct.

We may define an object for the abstract data type *Sequence_of_Item* as follows

```
OBJ Sequence_of_Item/Item
SORTS seq
OPS
  ˜: → seq
  ˆ: item seq → seq
JBO
```

The object *Sequence_of_Item* defines the sort *seq* with operators ˜ and ˆ. The semantics of an OBJ object is defined to be the initial algebra on the signature denoted by the sort and operator declarations.[12] In the case of the object *Sequence_of_Item* this means that the set of values (carrier) associated with the sort *seq* is isomorphic to the set of well-formed terms generated from the empty sequence operator, ˜, and left append ˆ, that is {˜,ˆ(*i*1,˜),ˆ(*i*2,˜),ˆ(*i*3,˜),ˆ(*i*1,ˆ(*i*1,˜)),...}).

The scope of operator names is governed by the rules given above for the use of sort identifiers.

Built-in objects

OBJ has a built-in object TRUTH which defines the sort BOOL with distinguished constants T (true) and F (false). The object TRUTH is implicitly accessible from any user-defined object.

```
OBJ TRUTH
SORTS BOOL
OPS
  T,F: → BOOL
JBO
```

In addition, UMIST OBJ provides a built-in implementation of the sort *nat*, with operations 0 and *succ* which have their usual interpretation over the natural numbers. The implementation corresponds to the object:

```
OBJ NATURAL
SORTS nat
OPS
  0: → nat
  succ: nat → nat
JBO
```

As with the TRUTH object above, access to NATURAL is implicitly given to all user-defined objects. The implementation allows natural numbers in canonical form (i.e. *succ*(*succ*(. . . . . . (*succ*(0)) . .)) to be written in standard decimal notation in equations and expressions, and prints them as such in dialogues with the user.

Equations

When specifying the behaviour of operators, it is usually necessary to provide definitions or axioms which constrain the allowable interpretations of the meaning. In an OBJ object, equations are used to define those relationships (identities) that must exist between values denoted by the constituent expressions involving operators.

Syntactically the equations of an object follow the keyword EQNS. The equations may involve typed (i.e. sort-constrained) implicitly universally quantified variables. Such variables are declared in advance to assist type checking. The variables employed in a particular equation are assumed to be universally quantified over the whole equation.

```
⟨var-declaration-list⟩ :: = ⟨var-decl⟩ {⟨var-decl⟩}
⟨var-decl⟩ :: = ⟨var-id⟩ {','⟨var-id⟩} ':' ⟨sort⟩
⟨var-id⟩ :: = ⟨id⟩

⟨equation⟩ :: = '('⟨op-exp⟩ = ⟨expression⟩
               ['IF ⟨bool-exp⟩]')'

⟨expression⟩ :: = ⟨var-id⟩ | ⟨op⟩ ['('⟨exp-list⟩')']
                 | '('⟨expression⟩')'
                 | ⟨expression⟩ '= =' ⟨expression⟩

⟨exp-list⟩ :: = ⟨expression⟩ {','⟨expression⟩}
```

For example, while the usual boolean operators are not declared in the built-in object TRUTH, they may be simply defined as follows:

```
OBJ Boolean
OPS
  not: BOOL → BOOL
  and,or: BOOL BOOL → BOOL
VARS
  a: BOOL
EQNS
  (not(T) = F)
  (not(F) = T)
  (and (a,T) = a)
  (and (a,F) = F)
  (and (T,a) = a)
  (and (F,a) = F)
  (or (a,T) = T)
  (or (a,F) = a)
  (or (T,a) = T)
  (or (F,a) = a)
JBO
```

The equations of object Boolean correspond to the usual definition of the boolean connectives by means of truth tables.

As a second example, the sequence data type is made more interesting by the introduction of a sequence append operator, *app*. The behaviour of *app* is defined by identities on terms formed by the application of *app* to all well-formed terms of sort *seq*, generated by the operators ˆ and ˜.

```
OBJ Sequence_with_Append/Item
SORTS seq
OPS
  ˜: → seq
  ˆ: item seq → seq
  app: seq seq → seq
VARS
  i: item
  s,s': seq
EQNS
  (app(˜,s) = s)
  (app(ˆ(i,s),s') = ˆ(i,app(s,s')))
JBO
```

The two equations define the operator *app* in a manner familiar to functional programmers. Note that OBJ's use of pattern matching, similar to that used in Hope,[2] reduces the need for selector operators such as *head* and *tail*, which are used heavily in languages such as Lisp. Each equation in an object determines that for any consistent substitution to the variables the values of the expressions on the left-hand side and right-hand side of the '=' are equal. Any two ground expressions (i.e. expressions which do not contain variables) which are not made equal using normal equality reasoning are considered unequal in the intended semantics.

OBJ also permits equations to be conditioned by a boolean expression, giving conditional equations of the general form ($l = r$ IF c). The equation $l = r$ holds under any acceptable substitution of terms for variables (of the appropriate sort) for which the boolean expression c is true (i.e. equal to T). As an example of the use of conditional equations, we may define a membership test operation, *isin?*, on the sequence data type:

```
OBJ
  Membership_Test/Sequence_with_Append Boolean
OPS
  isin?: item seq → BOOL
VARS
  i,j: item s: seq
EQNS
  (isin?(i,˜) = F)
  (isin?(i,ˆ(i,s)) = T)
  (isin?(i,ˆ(j,s)) = isin?(i,s) IF not(i = = j))
JBO
```

The condition part of the third equation ensures that the identity only holds when the first item of the sequence differs from the item being tested. Consequently, the second and third equations do not 'overlap'. That is, the respective identities hold in mutually disjoint contexts, so they can never be simultaneously applicable to the same expression.

The Equality Operator '= ='

As each sort, *s*, is introduced, it is automatically equipped with a boolean-valued equality operator, with infix syntax:

```
_ = = _: s s → BOOL
```

where underscores '_' act as placeholders for expression values. For two *s*-sorted terms, A, B, $A = = B$ is true (T) if the equality of A and B follows from the equations in the specification, and false (F) otherwise.

3. ENRICHMENTS AND HIERARCHIES OF DATA TYPES

An operator such as *app* may be defined as an enrichment of the existing sort *seq*, introduced in the original object *Sequence_of_Item* as follows, thus avoiding the need to redeclare existing sorts and operators:

```
OBJ Sequence_of_Item_2/Sequence_of_Item
OPS
  app: seq seq → seq
VARS
  i: item
  s,s': seq
EQNS
  (app(˜,s) = s)
  (app(ˆ(i,s),s') = ˆ(i,app(s,s')))
JBO
```

An OBJ specification constitutes an environment of modules with a hierarchical or acyclic graph structure. The structure is expressed and enforced by means of the used objects list. An object A can reference another object B iff B is in A's list of used objects, or B can be referenced by one of the objects used by A. The objects *Membership_Test*, *Sequence_with_Append*, *Item*, *Boolean*, together with the built-in objects TRUTH and NATURAL, form a single environment with structure shown in Fig. 1.

The enrichment of existing data types by the addition of new objects which use them is the principal specification-structuring mechanism available in UMIST OBJ.

$$\begin{tikzcd}
 & & \text{Membership\_Test} \arrow[dl] \arrow[ddr] & \\
 & \text{Sequence\_with\_Append} \arrow[d] & & \\
 & \text{Item} \arrow[dl] \arrow[dr] & & \text{Boolean} \arrow[dl] \\
\text{NATURAL} & & \text{TRUTH} &
\end{tikzcd}$$

Figure 1.

In general, every effort should be made to avoid the introduction of an object which corrupts the meaning of an existing object. Corruption of an existing sort can be caused by the introduction of operators whose meaning gives rise to new, previously unnameable values of the existing sort, or the introduction of equations which cause previously distinguishable values of the existing sort to become identified.

4. SEMANTICS OF ALGEBRAIC SPECIFICATIONS

4.1 Denotational semantics

The denotation of an OBJ object (resp. specification) is a many-sorted algebra: a collection of sets of values together with functions among them. The sorts (S) and operators (W) declared in an object constitute its 'signature', Σ, and a specification is considered to be a pair (Σ, E), where E is a set of Σ-equations. The denotation of an object is taken from the class of Σ-algebras, those many-sorted algebras having a carrier for each sort in S, and a (total) function for each operator in W. In general there exists a collection of algebras that, together with appropriate interpretations of the operator symbols, validate the set of equations in a specification. These constitute the 'variety' of models of an equational specification. The initial algebra approach to semantics uses the unique (up to isomorphism) 'most representative' algebra which satisfies the equations.[7,12] The sets of values of the initial model are obtained by first constructing sets of abstract values analogous to the Herbrand Universe of first-order predicate logic.[25] The equations generate a finest congruence relation used to quotient this set of values. The operations are interpreted as functions among the resulting sets that respect the signature and equations in the specification.

4.2 Axiomatic semantics

An axiomatic semantics can be given for UMIST OBJ by considering it as a form of logic programming language based on the simple and commonplace logic of (conditional) equations. The well-formed formulas of equational logic are equations defined as pairs of well-formed terms constructed from the operations declared in a signature and a set of variable identifiers, and written $M = N$. An equation $M = N$ is provable from a set of equations $E, E \vdash M = N$, if $M = N$ is in E or $M = N$ is obtained from E by a finite sequence of:

(i) $t = t$
(ii) if $s = t$ then $t = s$
(iii) if $r = s$ and $s = t$ then $r = t$
(iv) if $si = ti$, $1 < = i < = n$ then $f(s1, \ldots, sn) = f(t1, \ldots, tn)$ for any operator, f, of suitable arity.

The set of equations provable from a set of equations E is called the equational theory $=(E)$. Birkhoffs theorem[26] states that an equation is valid in **all** models of a set of equations, E, if and only if that equation is provable from E.

The adoption of initial algebra semantics for an OBJ specification causes equational logic to become incomplete for the associated equational theory. The restriction to 'nameable' models, in which any value in the carrier of a sort must be nameable by the operators in the specification, means that some inductive principle must be used to establish many interesting properties of a specification. Furthermore, the initial model uses the finest congruence generated by the equations in a specification, with the consequence that whenever an identity cannot be established between two ground terms by equational reasoning, those terms are considered unequal in the initial model. It is this semantics for equality which is implemented by the BOOL-valued equality operator, = =.

4.3 Operational semantics

An operational semantics is associated with an OBJ specification by treating each equation as an ordered pair,

left hand side → right hand side

and used as a rule for replacing one term by another.[16] A computation using rewrite rules produces a sequence $M, M1, M2 \ldots$ of expressions by repeatedly replacing instances of left-hand sides of rules by their corresponding right-hand sides until N is obtained, which contains no instance of any left-hand side. For example, the following rules derived from the equations in *Sequence_with_Append*

app(˜,*s*) → *s* (i)
app(ˆ(*i*,*s*),*s*')→ ˆ(*i*,*app*(*s*,*s*')) (ii)

can be used to rewrite the term

app(ˆ(*i*1,ˆ(*i*2,˜)),ˆ(*i*3,˜))

by first rewriting '*app*(ˆ(*i*1,ˆ(*i*2,˜)),ˆ(*i*3,˜))' to 'ˆ(*i*1,*app*(ˆ(*i*2,˜),ˆ(*i*3,˜)))' using (ii), next rewriting the resulting subterm '*app*(ˆ(*i*2,˜),ˆ(*i*3,˜))' using (ii) to yield 'ˆ(i1,ˆ(i2,*app*(˜,ˆ(*i*3,˜))))', and finally rewriting the subterm '*app*(˜,ˆ(*i*3,˜))' to 'ˆ(*i*3,˜)' using (i), to give the term 'ˆ(*i*1,ˆ(*i*2,ˆ(*i*3,˜)))', which cannot be further rewritten using (i) and (ii).

Under certain mild constraints, called finite convergence,[16] term rewriting provides an effective decision procedure for an equational theory presented by a set of (conditional) equations. The conditions for finite convergence are that:

(i) every terminating sequence of rewrites from some expression M stops at a unique minimal form M^* (referred to as the Church–Rosser property for recursive functions);
(ii) every sequence of rewrites from some expression M terminates after a finite number of steps, referred to as the finite termination property.

Under these conditions a ground equation $M = N$ can be proved to be a consequence of a set of equations E by exhaustively rewriting M and N and checking that M^* is identical to N^*.

In general it is undecidable whether an arbitrary set of rewrite rules is finitely convergent. In practice, it is relatively straightforward to ensure that a set of rewrite rules under construction have the necessary properties. Convergence can be achieved by strictly avoiding pairs of equations with 'overlapping' left-hand sides. Where conditional equations are involved the situation is more complicated, since the left-hand sides of a pair of equations may overlap syntactically, in which case the conditions on the two equations must be forced to denote mutually disjoint predicates, otherwise term rewriting with this pair of equations will result in a non-deterministic computation. Finite termination can be achieved by ensuring that replacing an instance of the left-hand side of an equation by the right-hand side results in an expression which is in some sense 'smaller'. It must be ensured that there is no infinite chain of expressions, each smaller than the next (cf. well-founded set). A very useful guide is to base all recursive definitions on primitive recursive schemata. Discussions of such an approach can be found in chapter 2 of Ref. 3, where it is referred to as structural recursion, and in Ref. 1, where it is related to the unique homomorphism property of initial algebra semantics.

5. EXPRESSION EVALUATION – THE RUN COMMAND

Before looking at a simple example of expression evaluation we will introduce one more object to our environment. This object introduces an operator which tests a sequence for the occurrence of duplicate items:

```
OBJ Duplicate_Test/Membership_Test
OPS
  nodups:seq → BOOL
VARS
  i:item s:seq
EQNS
  (nodups (ˆ) = T)
  (nodups(ˆ(i,s)) = nodups (s) IF not(isin?(i,s)))
  (nodups(ˆ(i,s)) = F IF isin?(i,s))
JBO
```

The evaluation of an expression is prompted by a ⟨*run command*⟩ which exhaustively uses a set of (conditional) equations as (conditional) rewrite rules.

⟨*run-command*⟩ :: = RUN ⟨*expression*⟩ NUR

The result of a run command is a message of form

AS ⟨*sortname*⟩:⟨*expression*⟩

which returns the canonical form of the expression together with its sort. Thus for example:

RUN *nodups* (ˆ(*i*7,ˆ(*i*3,ˆ(*i*2,ˆ(*i*6,ˆ(*i*2,ˆ(*i*9,ˆ))))))) NUR

results in

AS BOOL:F

and

RUN *isin*?(*i*2,ˆ(*i*7,ˆ(*i*3,ˆ(*i*2,ˆ(*i*6,ˆ(*i*2,ˆ(*i*9,ˆ))))))) NUR.

results in

AS BOOL:T

and

RUN *app*(ˆ(*i*2,ˆ(*i*7,ˆ(*i*3,ˆ))),ˆ(*i*6,ˆ(*i*2,ˆ(*i*9,ˆ)))) NUR

results in

AS *seq*: ˆ(*i*2,ˆ(*i*7,ˆ(*i*3,ˆ(*i*6,ˆ(*i*2,ˆ(*i*9,ˆ))))))

The equality operator, = =, may also be used in ⟨*run-commands*⟩

RUN *app*(ˆ(*i*2,ˆ(*i*7,ˆ)),ˆ(*i*2,ˆ)) = = ˆ(*i*2,ˆ(*i*7,ˆ(*i*2,ˆ))) NUR

produces

AS BOOL:T

The execution of OBJ specifications in this way provides a means of design-time testing the behaviour defined by abstract specifications of data types, and abstract applicative programs modelling the computation to be produced by some algorithm.

6. AN EXAMPLE USE OF UMIST OBJ

UMIST OBJ supports the production of formal specifications and design-time testing in the production of imperative and functional programs. It is useful when formulating the domain of discourse for specifications. The construction and evaluation of OBJ specifications can enhance our understanding of a design problem when, as is usually the case, we do not know what would constitute a reasonable formal requirements specification. The system permits the user to explore the theory underlying a design problem and its associated solutions at a level of abstraction motivated by his/her understanding of the problem domain, unobscured by as yet irrelevant detail demanded by the implementation technology. Constructing and exploring abstract executable specifications allows the designer to gain knowledge about the task at hand in a cheaper and more flexible manner than experimental programming, or 'prototyping', in a conventional programming language. Design-time testing, although incomplete, is a cost-effective way of increasing confidence in the correctness of software.[9,10] In the next section, we illustrate such a use of OBJ by a brief exploration of a topical problem. We also present a brief description of a number of non-trivial applications.

6.1 A simple program-component database

Consider the design of a simple database to hold Ada-like program components, which is to be accessed and updated by components of a development environment, such as a structure editor and Ada compiler. A component is named and comprises

an *interface* part

and

an *implementation* part.

The *interface* describes the visible part of a component, that is the declarations which are available to the user of the component. For example a stack component might have *interface*

```
interface STACK is
  type stack is private;
  procedure init(var s:stack);
  procedure pop(var s: stack; var i:item);
  procedure push(i:item; var s:stack);
  .
  .
end of interface
```

The implementation contains the code for the types, procedures and functions contained in the corresponding interface. Thus an implementation corresponding to stack might be

```
implementation STACK is
  imports sequence;
  type stack = sequence of item;
  procedure init(var s:stack); begin initialise(s) end;
  procedure push (i:item;var s:stack); begin right-
    append(s,i) end;
  procedure pop(var s: stack; var i:item);
  .
  .
end of implementation
```

Both implementations and interfaces may access other components by listing their names in an imports statement. Thus the above implementation of stack imports the sequence component. In general there may be more than one implementation for a given interface, since usually there are many ways of implementing a given data type. For the purposes of this simple example we ignore this possibility.

We can formalize the ideas of interface and implementation as follows.

```
OBJ Name_List
SORTS name name_list
OPS
  ˜: → name_list
  ˆ:name name_list → name_list
JBO

OBJ Part_Type
SORTS part_type
OPS
  interface,impln : → part_type
JBO

OBJ Part/Name_List Part_Type
SORTS part
OPS
  %:name part_type name_list → part
  nameof:part → name
  kind:part → part_type
  imports:part → name_list
VARS
  comp_name:name comp_kind:part_type
  imports_list:name_list
EQNS
  (nameof(%(comp_name,comp_kind,imports_list)) =
    comp_name)
  (kind(%(comp_name,comp_kind,imports_lists)) =
    comp_kind)
  (imports(%(comp_name,comp_kind,imports_list)) =
    imports_list)
JBO
```

OBJ is used to provide abstract syntax for the semantically important notions of a part. We can use this level of abstraction since we do not need to elaborate the procedures and types in the interface. Similarly we are not concerned with the actual code that constitutes the implementation body.

We need a mechanism for storing and retrieving program parts. The following object allows us to capture the notion of storing and retrieving interfaces and implementations associated with some component name.

```
OBJ Database/Part Boolean
SORTS database
OPS
  @: → database
  put:part database → database
  getinterface:name database → part
  getimpln:name database → part
VARS
  n:name p,p':part db: database
EQNS
  (getinterface(n,put(p,db)) = p  IF  and  (nameof(p)
    = = n,kind(p) = = interface))
  (getinterface(n,put(p,db)) = getinterface(n,db)    IF
    or(not(nameof(p) = = n),not(kind(p) = = inter-
    face)))
  (getimpln(n,put(p,db)) = p  IF  and(nameof(p) = =
    n,kind(p) = = impln))
  (getimpln(n,put(p,db)) = getimpln(n,db)  IF  or(not-
    (nameof(p) = = n),not(kind(p) = = impln)))
JBO
```

Thus '@' corresponds to the initialised empty database and 'put' inserts a part into an existing database value. The operation *getinterface* (*resp. getimpln*) may be used to retrieve the latest interface (*resp. implementation*) with a particular name, to have been put into a database value. Another useful database operation performs a check on the presence of an implementation for a named component. The requirement for such an operator can be given informally as:

```
(for_all n:name, db:database).
  (impln_exists?(n,db) ↔
  (exists p:part).(nameof(p) = = n
  AND
  kind(p) = = impln AND in?(p,db)))
```

where *in?* is the membership test on database values.

We can construct the test *impln_exists?* as a BOOL-valued operator in a number of ways. The following definition illustrates the use of structural recursion.

```
OBJ Implementation_Exists/Database
OPS
  impln_exists? :name database → BOOL
VARS
  n:name p:part db:database imports_list:name_list
EQNS
  Impln_exists?(n,@) = F)
  (impln_exists?(n,put(%(n,impln,imports_list),db)) =
    T)
  (impln_exists?(n,put(p,db)) = impln_exists?(n,db)
    IF or(not(nameof(p) = = n),
        not(kind( p) = = impln)))
JBO
```

It is left as an unproven verification condition that the object for *impln_exists?* meets its requirement.

One application for such a database would be to act as the back end of a component compiler. Before a program can be compiled into object code, not only must an implementation exist for that component but also for the transitive closure of all components that it imports. If we assume no circularities in the imports relation then the requirement for an operator to check whether a named component can be compiled can be stated as:

```
(for_all n:name, db:database).
  (can_compile?(n,db) ↔
    (impln_exists?(n,db) AND
    (for_all n' : name)
    (isin?(n',imports(n,db)) → can_compile?(n',db))))
```

where *isin?* is the membership test on *name_list's*.

One obvious way to construct the required behaviour for *can_compile?* is given below, where the operator is defined by mutual recursion with the test.

```
OBJ Can_Compile/Implementation_Exists
OPS
  can_compile? : name database → BOOL
  'can_compile? : name_list database → BOOL
VARS
  n : name p : part db : database nl : name_list
EQNS
  (can_compile?(n,db) =
     'can_compile?(imports(getimpln(n,db),db)
     IF impln_exists?(n,db))
  (can_compile?(n,db) = F
     IF not(impln_exists?(n,db)))
  ('can_compile?(~,db) = T)
  ('can_compile?(~(n,nl),db) =
     and(can_compile?(n,db), 'can_compile?(nl,db)))
JBO
```

This brief example illustrates the use of OBJ to support the specification of the behaviour of systems. The axiomatic definitions do not rely upon any *a priori* notion of flow of control. In the above example we have a functional description of a simple database, in which there is no notion beyond that of the mapping between input values and results. We need richer logics to capture detailed structural and operational properties, such as sequential, concurrent or distributed implementations.[15]

6.2 Further applications of UMIST OBJ

The UMIST OBJ system has been successfully used in a number of non-trivial software engineering and other problems. Below, we present a brief survey of these applications.

Coleman *et al.* discuss the design of a rewrite rule engine from algebraic specifications and also present the associated design methodology in detail.[4] Both methodology and software have been extensively used during the development of the UMIST OBJ system. Earley's algorithm[6, 17] has been specified and implemented, as well as the Knuth–Bendix completion procedure.[18] A lazy associative-commutative pattern matcher has also been implemented using the system.[19]

Other applications include a configuration management system implemented in Ada,[20] specification of protocols,[21] graphics software (GKS),[5] concurrent systems, programming language semantics and specification and verification of hardware.[22]

7. THE UMIST OBJ IMPLEMENTATION

The UMIST OBJ system comprises two modules, a compiler and an interpreter, interfaced by an intermediate environment file.

7.1 The compiler

The compiler takes its input from a file containing OBJ source text, previously prepared using a text editor. It performs lexical and syntactic analysis and type checking on its input. It compiles the equations of each object to corresponding rewrite rules encoded in an internal format shared with the interpreter. The module structure, and acyclic graph structure expressed through the access relation between objects, are used to apply the scoping rules to the use of declared sorts and operators. This structure is not preserved in the set of rewrite rules generated by the compiler, the latter assuming a 'flattened' list form.

The language subset described here is characterizable by an LL(1) context-free grammar. Consequently, after lexical analysis, syntactic analysis is performed using a recursive descent parser. The syntax used for recognizing expressions occurring in equations is liberal, since the expression language is extended by each sort and operator declaration. The signature information of each object, which declares sorts and operators, is maintained in a compile-time symbol table, and used to check the well-formedness of expressions occurring in each equation. This involves the strong type checking of each occurrence of constant and variable identifiers, the number and sort of each actual argument and the target sort of each operator application.

In the version of the compiler described here, no action is taken to recover from syntactic or semantic (type consistency) errors. Such errors are reported immediately on encounter with appropriate diagnostic messages. On successful compilation, the compiler generates the interface file which represents the environment in which expressions belonging to the user-defined syntax will be evaluated.

Successful compilation causes the resulting list of rewrite rules to be written out to an environment file, together with a précis of the declarations to be used for typechecking expressions in RUN commands and formatting answers in the interpreter. The internal format for expressions and equations, shared by the compiler and interpreter, encodes all external identifiers as integer values. User-defined identifiers are encoded by the index of their occurrence in the compilers symbol table.

7.2 The interpreter

The interpreter provides an interactive environment for executing sets of rewrite rules to evaluate expressions. The environment file is accessed by the interpreter to initialise its internal state with the necessary information to undertake expression evaluation, in response to user-given RUN commands, using the rewrite rules. Other commands are available to the user, to inspect the rewrite rules currently in use, set the level of tracing and so on. The interpreter can be directed to take its command input from a specified file, rather than interactively, and to send its results, trace output and so on to a named file.

The principal component of the interpreter is an expression evaluator in the form of a term-rewriting engine. It provides the computation rule and pattern-matching machinery necessary to execute the rules compiled from a source specification against user-defined expressions. The behaviour of the term-rewriting engine is elaborated in Ref. 4, which also explains how OBJ specifications were put to use during its development.

7.3 The interpreter command language

The interpreter makes available a number of interactive commands, in addition to the RUN command itself. The computation rule used by the interpreter can be switched between 'top-down' and 'bottom-up' modes of evaluation. Top-down evaluation attempts to perform reductions at the outermost level of operator application in the expression under evaluation. Only when rewriting is not possible at that level is an attempt made to reduce the arguments. Attention reverts to the top level once a reduction has been noted in any argument position. In contrast, in bottom-up mode, nested subexpressions corresponding to the arguments of an operator application are always exhaustively rewritten before rules involving the operator are used at the outer level. The choice of computation rule can generally affect the performance of the interpreter on particular problems. However, for finitely convergent systems of rewrite rules it has no effect on the completeness of the term-rewriting decision procedure.

The user can also specify whether the interpreter should employ sharing of commonly occurring subexpressions. The use of sharing of common substructures can increase both space and time efficiency of a computation. On the one hand, the storage requirement of the program is obviously reduced. On the other hand, the interpreter performs a smaller amount of copying of terms, and multiple evaluations of the same expression are avoided. The latter property can have a marked effect in the case of conditional equations, where the evaluation of the condition part and the right-hand side of the equation involve the reduction of non-trivial common subexpressions.

In summary, the following interactive commands are available:

option	meaning
run ⟨*expression*⟩ *nur*	Evaluate ⟨*expression*⟩ using the rewrite system loaded on interpreter initialisation.
evaln bu | *td*	Select *bottom up* (*resp. top down*) evaluation mode (default is *bottom up*).
evaln share | *noshare*	Select sharing (*resp. no sharing*) of common subexpressions during evaluation (default is *share*).
tr ⟨*trace-level*⟩	Trace the evaluation of expressions level 0: no tracing (default) level 1: trace evaluation of the original expression only level 2: trace evaluation including conditions.
show all | ⟨*operator*⟩	Display all the rules in the environment, or just those defining ⟨*operators*⟩.
time	Display the date, time and environment name.
⟨*text*⟩	Treats ⟨*text*⟩ as comments.
help | *?*	List information about the interpreter command language.
exit	Leave the interpreter and terminate the session.

8. CURRENT STATUS, AVAILABILITY AND FUTURE WORK

The pre-release version of the UMIST OBJ system described here was developed in Pascal on an ICL PERQ running under the PNX operating system and subsequently on a DEC VAX 11/750 running 4.2 BSD UNIX. The system is also available on a multitude of machines and operating systems. This version has been widely distributed to academia and industry in the UK abroad.

Further work undertaken resulted in the first full release of the system.[23] Release 1.0 features:

(i) A parser based on Earley's Algorithm[6, 17] to allow parsing of operators with mixfix syntax.

(ii) An enhanced term-rewriting system. In this release operators may be given attributes to express properties which are not conveniently defined by equations. Although these properties are closely associated with the syntax of a given operator, they are also in part semantic properties. Binary infix operators can be declared to be either associative or commutative or associative and commutative. It is also possible to give an operator a left and right identity element. For instance, set union is associative and commutative and has the empty set as an identity element:

_U_: *set set* → *set* (ASSOC COMM ID: *phi*)

The implementation of these properties requires special mechanisms. The ASSOC attribute is needed to enable the correct parsing and deparsing of unparenthesised terms as well as allowing the pattern matching of terms with arbitrary arity. All the other attributes require enhancements to the interpreter. For example commutativity is implemented by using both orientations of a term during pattern matching. Release 1.0 includes a lazy pattern matcher as well as a specialised Knuth–Bendix completion procedure for handling identity.[24]

(iii) Integers are handled using the underlying Pascal arithmetic.

This software is currently available under commercial licence.

9. CONCLUSION

This concludes our introduction to the elements of OBJ supported by the existing release of the UMIST OBJ system. Although lacking in sophistication compared to implementations of larger subsets of the language, the system is powerful and robust enough to provide support for the development of non-trivial software.

The full potential of formal notations such as OBJ will be best realised in integrated software engineering environments, comprising syntax-directed structure editors, language processors, consistency checkers such as the Knuth–Bendix algorithm,[16] validation and verification tools, organised around a central module database. Finally, and perhaps most vitally, effective software engineering based on formal specifications requires the development of realistic methodologies derived from models of the software design process.[3, 15, 17]

Acknowledgement

The software described in this paper was developed while D. Coleman and R. M. Gallimore were lecturers in the

Computation Department, UMIST. The work reported here was supported by ALVEY/SERC grant no. GR/C/31655.

We would like to thank Joseph Goguen for his assistance and encouragement, and Cliff Jones for his friendship and support.

The compiler of the UMIST OBJ system was implemented by F. H. Ali.

REFERENCES

1. R. M. Burstall, Inductively defined functions. *Proceedings of the International Joint TAPSOFT Conference, Berlin, March 1985*, vol. 1, edited H. Ehrig, C. Floyd, C. Nivat and J. Thatcher (LNCS 185, pp. 92–96). Springer, Heidelberg (1985).
2. R. M. Burstall, D. MacQueen and D. Sanella, HOPE: an experimental applicative language. LISP Conference, Stanford University (1980).
3. D. Coleman and R. M. Gallimore, Software engineering using executable specifications. Notes from a course given at the Computation Department, UMIST (1985).
4. D. Coleman, R. M. Gallimore and V. Stavridou, The design of a rewrite rule interpreter from algebraic specifications. *IEE Software Engineering Journal*, **2** (4) (1987).
5. D. A. Duce and E. V. C. Fielding, Formal specification – a comparison of two techniques. *The Computer Journal*, **30** (4), 316–327 (1987).
6. J. Earley, An efficient context-free parsing algorithm. *Communications of ACM*, **13** (2) (1970).
7. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification. 1. Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science, vol. 6. Springer, Berlin (1985).
8. K. Futatsugi, J. A. Goguen, J-P. Jouannaud and J. Meseguer, Principles of OBJ2. *Proceedings of the 1985 Symposium on Principles of Programming Languages, ACM* (1985).
9. N. Gehani, Specifications: formal and informal – a case study. *Software Practice and Experience* **12**, 433–444 (1982).
10. C. P. Gerrard, D. Coleman and R. M. Gallimore, Formal specification and design time testing. To appear in *IEEE Transactions on Software Engineering*.
11. J. A. Goguen, Parameterized programming. *IEEE Transactions on Software Engineering* **SE-10** (5), 528–543 (1984).
12. J. A. Goguen and J. Meseguer, *An Initiality Primer*. SRI International, Computer Science Laboratory, Menlo Park, California (1983).
13. J. V. Guttag, Abstract data types and the development of data structures. *Communications of ACM* **20** (6) (1977).
14. J. V. Guttag, Notes on type abstraction (Version 2). *IEEE Transactions on Software Engineering* **SE-6** (1) (1980).
15. J. V. Guttag, J. J. Horning and J. M. Wing, The larch family of specification languages. *IEEE Software* **2** (5) (1985).
16. G. Huet and D. Oppen, Equations and rewrite rules: a survey. In *Formal Language Theory: Perspectives and Open Problems*, edited R. Book. Academic Press, London (1980).
17. C. B. Jones, *Software Development: A Rigorous Approach*. Prentice-Hall International, Englewood Cliffs, N.J. (1980).
18. D. Knuth and P. Bendix, Simple word problems in Universal algebras. In *Computational Problems in Abstract Algebra*, edited J. Leech, pp. 263–297. Pergamon Press, Oxford (1970).
19. D. Coleman, R. M. Gallimore and V. Stavridou, *Term Rewriting with Attributes in UMIST OBJ* (in preparation).
20. C. P. Gerrard, *The Specification and Controlled Implementation of a Configuration Management Tool Using OBJ and Ada*. Gerrard Software, Venture House, Cross St, Macclesfield SK11 7PG.
21. M. Diss, *Formal Specification of the Transport Layer Protocol using LOTOS/ACT ONE*. Report No. 72/86/R219U, Plessey Research Roke Manor Ltd, Roke Manor, Romsey, Hampshire, SO51 0ZN (May 1986).
22. V. Stavridou, *Specifying in OBJ, Verifying in REVE and Some Ideas About Time* (in preparation).
23. C. D. Walter, R. M. Gallimore, D. Coleman and V. Stavridou, *UMIST OBJ Manual Version 1.0*. Computation Department, UMIST, Manchester M60 1QD (1986).
24. D. Knuth and P. Bendix, Simple word problems in universal algebras. In *Computational problems in Abstract Algebra*, edited J. Leech, pp. 263–297. Pergamon Press, Oxford (1970).
25. Z. Manna, *Mathematical Theory of Computation*. McGraw-Hill, Maidenhead (1974).
26. G. Birkhoff, On the structure of abstract algebra. *Proceedings of the Cambridge Philosophical Society* **31**, 433–454 (1935).