SRC TR 89-46

Operational Relational Model - Implementation Through Specification

By

R. J. Cochrane

Operational Relational Model – Implementation Through Specification

Roberta Jo Cochrane

Fall 1988

1. Abstract

Our research evolves around the specification and implementation of the Operational Relational Model, (ORM), a model based on Update Dependencies [MARK1]. The intension and extension dimensions are explored through the construction of a self–describing system. The result, the implementation of ORM, will consist of a production system for specifying operations as an integral part of the conceptual schema of the database.

2. Introduction

Operational Relational Model, (ORM), is a data model that includes operation specification as an integral part of the conceptual schema design. ORM is an extension of the relational model which includes Update Dependencies as its operation specification language.

This paper serves as a summary of the ORM implementation through specification. It begins with a brief introduction to Update Dependencies followed by a discussion of the ORM implementation. The Appendices contain the complete specification of ORM minus the semantic actions for the control strucuture.

3. Overview of Update Dependencies

Update Dependencies is a formalism for specifying database operations for controlling updates [MARK1]. Similar to the concept of abstract data structures, this formalism provides an abstraction facility that allows the database designer to group together data operations. In this fashion, the knowledge required to maintain integrity constraints while performing an update request is encoded in the database. Update Dependencies is a unique approach to integrity constraint specification in that the user defines the correct evolution, (i.e. the correct transition from one valid state to another), rather than the valid states of the database.

The remainder of this section describes operation specification and execution in Update Dependencies as presented in [MARK1].

Each operation specification has the following form:

define_op <op> :- <dependency_1> .
:- <dependency_2> .
:- ... .
:- <dependency_N>.

where N >=1 and <op> is the name of the operation being defined. So, an operation is defined by one or more <dependencies>.

Each <dependency> is a condition followed by zero or more implied operation invocations. It is given by the form:

<cond> ,
<impl_op1> ,

...
<impl_opM>

where M >=0, <cond> is a condition, and <impl_op> is the invocation of an operation. Valid conditions are comprised of a subset of first order logic plus predicates for testing the existence of qualified tuples and the instantiation of predicates.

There are four types of operations: user, abstract, primitive, and I/O. User and abstract operations must be defined by an operation specification. Primitive and I/O operations are supplied primitive language constructs.

User operations allow the user to group together abstract operations, user operations, and conditions that must be satisfied before invoking a sequence of these operations.

Abstract operations allow the designer to group together conditions that must be satisfied and operations that must be performed when a relation is updated. There are four categories of these operations: insert, delete, modify, and get. The database designer should specify an operation from each of these categories for each relation in the database.

There are primitive operations for adding and removing tuples from relations, and one for obtaining a new surrogate value. Primitive operations are restricted to occur only within the definition of abstract operations. Furthermore, a primitive operation invocation over a relation, R1, is restricted to occur only within the definition of abstract operations defined over the same relation, R1. This strict hierarchy forces the definition and use of abstract operations to perform the primitive update operations on the database.

The I/O operations consist of read, write and break. Break is for temporarily suspending the system during an operation execution.

An operation execution succeeds when at least one of its <dependencies> succeeds. A <dependency> succeeds if the condition evaluates to true and all the <impl_ops> succeed. The execution of an operation is analogous to computing a query in logic programming. Each <dependency> can be thought of as a clause body where the corresponding clause head is defined as the <op> under which the dependency occurs.

The order in which <dependencies> are chosen for execution is nondeterministic and chosen at run–time. The evaluation of conditions and execution of implied operations is left–to–right. A closed world interpretation is assumed for the evaluation of conditions, (i.e. facts derived from the positive facts in the database are true, and facts that cannot be derived in such a fashion are assumed false).

Example

The following example is based on the suppliers–parts–shipments database, consisting of the following relations:

SUPP(S#:suppl_num, SNAME:suppl_name)
PART(P#:part_num, PNAME:part_name)
SHIP(S#:suppl_num, P#:part_num, QTY:integer)

There are referential integrity constraints from S# and P# in SHIP to S# in SUPP and P# in PART respectively. This example is an operation specification for the insertion of a shipment

that enforces referential integrity.

```
insert(SHIP(S#=S, P#=P, QTY=Q))
        :-  nonvar(S) and nonvar(P) and nonvar(Q) and
            SUPP(S#=S) and PART(P#=P),
            add(SHIP(S#=S, P#=P, QTY=Q)).
        :-  nonvar(S) and nonvar(P) and nonvar(Q) and
            not SUPP(S#=S),
            insert(SUPP(S#=S)),
            insert(SHIP(S#=S, P#=P, QTY=Q)).
        :-  nonvar(S) and nonvar(P) and nonvar(Q) and
            SUPP(S#=S) and not PART(P#=P),
            insert(PART(P#=P)),
            insert(SHIP(S#=S, P#=P, QTY=Q)).
        :-  var(S),
            write("Input supplier number"),
            read(S),
            insert(SHIP(S#=S, P#=P, QTY=Q)).
        :-  var(P),
            write("Input part number"),
            read(P),
            insert(SHIP(S#=S, P#=P, QTY=Q)).
        :-  var(Q),
            write("Input quantity"),
            read(Q),
            insert(SHIP(S#=S, P#=P, QTY=Q)).
```

One of the first three dependencies will apply when all the parameters needed to insert the tuple have been specified. One of the last three cases are executed if the corresponding attribute value has not been instantiated. In this case the user is prompted for the appropriate value, and recursion is used to insert the shipment with the new information.

The first dependency is the case when the values of the part number and supplier number satisfy the referential integrity constraint. In this case the indicated shipment is added to the relation SHIP.

The second dependency is the case when the supplier number does not satisfy the referential integrity constraint. In this case we have chosen to try to insert the supplier with the given supplier number into the database. If this insertion succeeds, recursion is used to attempt to insert the shipment again.

Similarly, the third dependency is the case when the part number does not satisfy the referential integrity constraint. Again, we have chosen to try to insert the part with the indicated supplier number into the database, followed by a recursive call to the insert shipment operation. The test to insure that the supplier number satisfies referential integrity is not needed, but added to avoid unnecessary computation.

4. Implementation of ORM Through Specification

The implementation of ORM consists of the specification of four parts utilizing four specification languages. The four parts consist of the syntax, meta–schema, semantic actions, and control structure. The specification languages employed are LEX, YACC, Update Dependencies, and the Relational Data Model.

The syntax of the ORM language (Appendix A) is defined by YACC and LEX specifications. YACC and LEX will automatically generate a parser for this language.

The ORM implementation includes a conceptual meta–schema that defines all schemata (Appendix B) and all operations on schemata that can be defined in terms of the Relational Data Model and Update Dependencies. This meta–schema is defined in terms of the Relational Data Model and Update Dependencies. Thus, the intension of the meta–schema can be stored in the extension of the meta–schema, (i.e. as part of the meta–schema data).

The meta–schema is a refinement of the meta–schema developed in [MARK2] that includes the explicit representation of the internal structure of conditions and expressions. The meta–schema is designed with the goal of representing every aspect of the definitions explicitly. Intuitively, such exact modeling will make queries and operations on the database expensive. However, this representation is flexible and advantageous for future research and development, such as enabling an update to the schema to be propagated to operations affected by the update.

We are currently in the process of specifying the semantic actions for the ORM language. Our goal is to utilize, as much as possible, the power of Update Dependencies for specifying and implementing these semantic actions. All data definition statments have been implemented as simple insertions on the relations in the meta–schema and are thus invocations of meta–schema operations (Appendix C).

We are currently investigating the difficult issue of specifying the operation execution in Update Dependencies. Our results will strongly affect the control structure, in that the control structure will consist of those parts of the operation execution which are not specified in Update Dependencies. If the search strategy and the unification algorithm are specified in Update Dependencies, these algorithms are modifiable within Update Dependencies and alternative strategies can be specified by the database designer. Furthermore, once it is clear how to specify the semantic actions for operation execution, we need to identify the necessary initial contents of the meta–schema.

We have found that the specification of the semantic actions in Update Dependencies has been greatly simplified by defining views on the meta–schema. We first defined the interpreter's view of the meta–schema and the abstract operations needed to perform updates on these views. The semantic actions consist of invocations of these abstract operations on the defined views.

5. Future Research

The specification and implementation of the Operational Relational Model raises many additional interesting questions:

- Can a theory be developed for proving correct evolution of the database based on this formalism?
- Is there a way to efficiently organize the dependency clauses in an operation specification to improve operation execution? Would it be of any benefit to specify post conditions for each dependency clause?
- Can we avoid tuple at a time access to the database, and how?
- Can an algebra be developed for specifying the combination of operation specifications, such as merging two operation specifications that are defined over the same operation?
- Many language constructs, such as iteration and alternation have a direct translation to Update Dependencies [MARK1]. How should these language constructs be supported by ORM?

6. Conclusion

Update Dependencies provides a formalism for operational database specification. Analysis indicates that this formalism is very powerful, and provides the basis for many more research issues [MARK1]. We are implementing an extension of the relational model, ORM, utilizing Update Dependencies as the operation specification language. The goal is to employ a very simple control structure for the model's language interpreter, specifying most of the semantic actions in Update Dependencies.

References

[MARK1] Mark, L., Roussopoulos, N., "Operational Specification of Update Dependencies", Department of Computer Science and Systems Research Center, University of Maryland, College Park, Maryland, Technical Report TR-87-37, 1987.

[MARK2] Mark, L., "Self-Describing Database Systems - Formalization and Realization", Department of Computer Science, University of Maryland, College Park, Maryland, Technical Report TR-1484, April 1985.

Appendix A – Syntax

The syntax of the specification language is defined as follows, where non–terminals are in capital letters:

```
10   PROG              ::=   OP_LIST

30   OP_LIST           ::=   OP_LIST INTERACT_OP
40                       |   INTERACT_OP

60   INTERACT_OP       ::=   DOM_DEF
70                       |   REL_DEF
80                       |   VIEW_DEF
90                       |   OPER_DEF
110                      |   ABSTR_OP_CALL.
120                      |   USER_OP_CALL.
130                      |   I_O_OP_CALL.

140  DOM_DEF           ::=   define_dom DOM_NAME_LIST:DOM_TYPE.

150  DOM_NAME_LIST     ::=   DOM_NAME_LIST, DOM_NAME
160                      |   DOM_NAME

170  DOM_TYPE          ::=   DOM_NAME
180                      |   PRIM_TYPE

190  PRIM_TYPE         ::=   integer
200                      |   char(int_lit)
210                      |   surrogate
220                      |   real

230  REL_DEF           ::=   define_rel REL_NAME(ATTR_DOM_LIST).

240  VIEW_DEF          ::=   define_view REL_NAME(ATTR_VAR_LIST) COND_ALT_SEQ

250  OPER_DEF          ::=   define_op ABSTR_OP_DEF
260                      |   define_op USER_OP_DEF

270  ABSTR_OP_DEF      ::=   get(REL_NAME(ATTR_VAR_LIST)) ABSTR_ALT_SEQ
280                      |   insert(REL_NAME(ATTR_VAR_LIST)) ABSTR_ALT_SEQ
290                      |   delete(REL_NAME(ATTR_VAR_LIST)) ABSTR_ALT_SEQ
300                      |   modify(REL_NAME(ATTR_VAR_LIST; ATTR_VAR_LIST))
                             ABSTR_ALT_SEQ

315  COND_ALT_SEQ      ::=   :– COND_ALT_SEQ COND.
317                      |   :– COND.

320  ABSTR_ALT_SEQ     ::=   :– ABSTR_ALT_SEQ ABSTR_ALT.
```

```
330                     |   :- ABSTR_ALT.

340   ABSTR_ALT         ::= COND, ABSTR_OP_SEQ
350                     |   COND

360   ABSTR_OP_SEQ      ::= ABSTR_OP_SEQ, ABSTR_IMPL_OP
370                     |   ABSTR_IMPL_OP

380   ABSTR_IMPL_OP     ::= ABSTR_OP
390                     |   PRIMIT_OP
400                     |   I_O_OP

410   ABSTR_OP          ::= get(REL_NAME(ATTR_EXPR_LIST))
420                     |   insert(REL_NAME(ATTR_EXPR_LIST))
430                     |   delete(REL_NAME(ATTR_EXPR_LIST))
440                     |   modify(REL_NAME(ATTR_EXPR_LIST; ATTR_EXPR_LIST))

450   PRIMIT_OP         ::= add(REL_NAME(ATTR_EXPR_LIST))
460                     |   remove(REL_NAME(ATTR_EXPR_LIST))
470                     |   new(DOM_TYPE , VARIABLE)

480   I_O_OP            ::= write(EXPR)
500                     |   read(VARIABLE)
510                     |   break

520   USER_OP_DEF       ::= OP_NAME(REL_NAME(ATTR_VAR_LIST)) USER_ALT_SEQ
530                     |   OP_NAME(REL_NAME(ATTR_VAR_LIST; ATTR_VAR_LIST))
                            USER_ALT_SEQ
535                     |   OP_NAME(ATTR_VAR_LIST)) USER_ALT_SEQ

540   USER_ALT_SEQ      ::= :- USER_ALT_SEQ USER_ALT.
550                     |   :- USER_ALT.

560   USER_ALT          ::= COND, USER_OP_SEQ
565                     |   COND

570   USER_OP_SEQ       ::= USER_OP_SEQ, USER_IMPL_OP
580                     |   USER_IMPL_OP

590   USER_IMPL_OP      ::= ABSTR_OP
600                     |   USER_OP
610                     |   I_O_OP

620   USER_OP           ::= OP_NAME(REL_NAME(ATTR_EXPR_LIST))
625                     |   OP_NAME(REL_NAME(ATTR_EXPR_LIST; ATTR_EXPR_LIST))
627                     |   OP_NAME(ATTR_EXPR_LIST))

630   COND              ::= (COND)
```

```
640                     |   not COND
650                     |   COND and COND
660                     |   REL_NAME(ATTR_EXPR_LIST)
680                     |   EXPR COMP_OP EXPR
690                     |   var(VARIABLE)
700                     |   nonvar(VARIABLE)
713                     |   true
716                     |   false

720  COMP_OP        ::=     <
721                     |   <=
722                     |   =
723                     |   <>
724                     |   >
725                     |   >=

730  ATTR_DOM_LIST  ::=     ATTR_DOM_LIST, ATTR_NAME:DOM_TYPE
740                     |   ATTR_NAME:DOM_TYPE

810  ATTR_EXPR_LIST ::=     ATTR_EXPR_LIST, ATTR_NAME=EXPR
820                     |   ATTR_NAME=EXPR
830                     |   epsilon

835  ATTR_VAR_LIST  ::=     ATTR_VAR_LIST, ATTR_NAME=VARIABLE
836                     |   ATTR_NAME=VARIABLE
837                     |   epsilon

848  EXPR           ::=     EXPR+EXPR
860                     |   EXPR-EXPR
870                     |   EXPR*EXPR
880                     |   EXPR/EXPR
890                     |   (EXPR)
900                     |   -EXPR
910                     |   CONST
925                     |   VARIABLE

950  REL_NAME       ::=     identifier

960  ATTR_NAME      ::=     identifier

970  DOM_NAME       ::=     identifier

980  OP_NAME        ::=     identifier

990  CONST          ::=     str_lit
1000                    |   int_lit
1010                    |   real_lit
```

1020	VARIABLE	::=	**identifier**
1030	ABSTR_OP_CALL	::=	**get**(REL_NAME(ATTR_EXPR_LIST))
1040		|	**insert**(REL_NAME(ATTR_EXPR_LIST))
1050		|	**delete**(REL_NAME(ATTR_EXPR_LIST))
1060		|	**modify**(REL_NAME(ATTR_EXPR_LIST; ATTR_EXPR_LIST))
1070	I_O_OP_CALL	::=	**write**(EXPR)
1080		|	**read**(VARIABLE)
1090		|	**break**
2000	USER_OP_CALL	::=	OP_NAME(REL_NAME(ATTR_EXPR_LIST))
2010		|	OP_NAME(REL_NAME(ATTR_EXPR_LIST; ATTR_EXPR_LIST))
2010		|	OP_NAME(ATTR_EXPR_LIST))

Appendix B – Meta Schema

The meta–schema for the Operational Data Model is presented in a graphical notation defined as follows:

Boxes represent relations, with attribute names in the interior of the subdivisions of the boxes, and the relation name on the exterior of the box near the box.

Solid Circles represent surrogate domains, with the domain name on the interior of the circle.

Dashed Circles represent lexical domains, with the domain name on the interior of the circle.

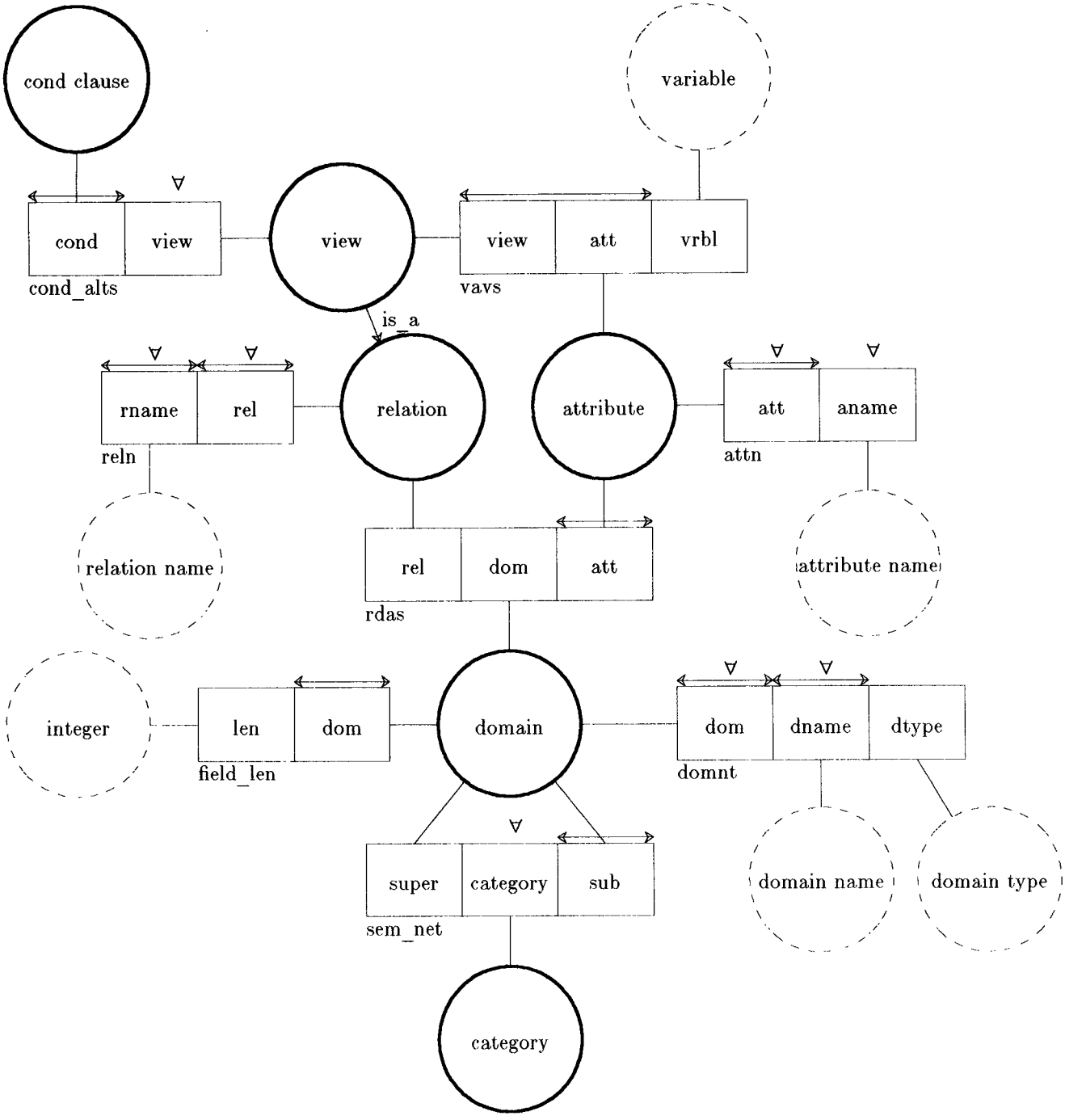
Lines between an attribute field of a relation and a domain indicates that the attribute takes its values from the domain.

Double headed lines over attribute fields indiciates that the attribute fields constitute a primary key for the relation.

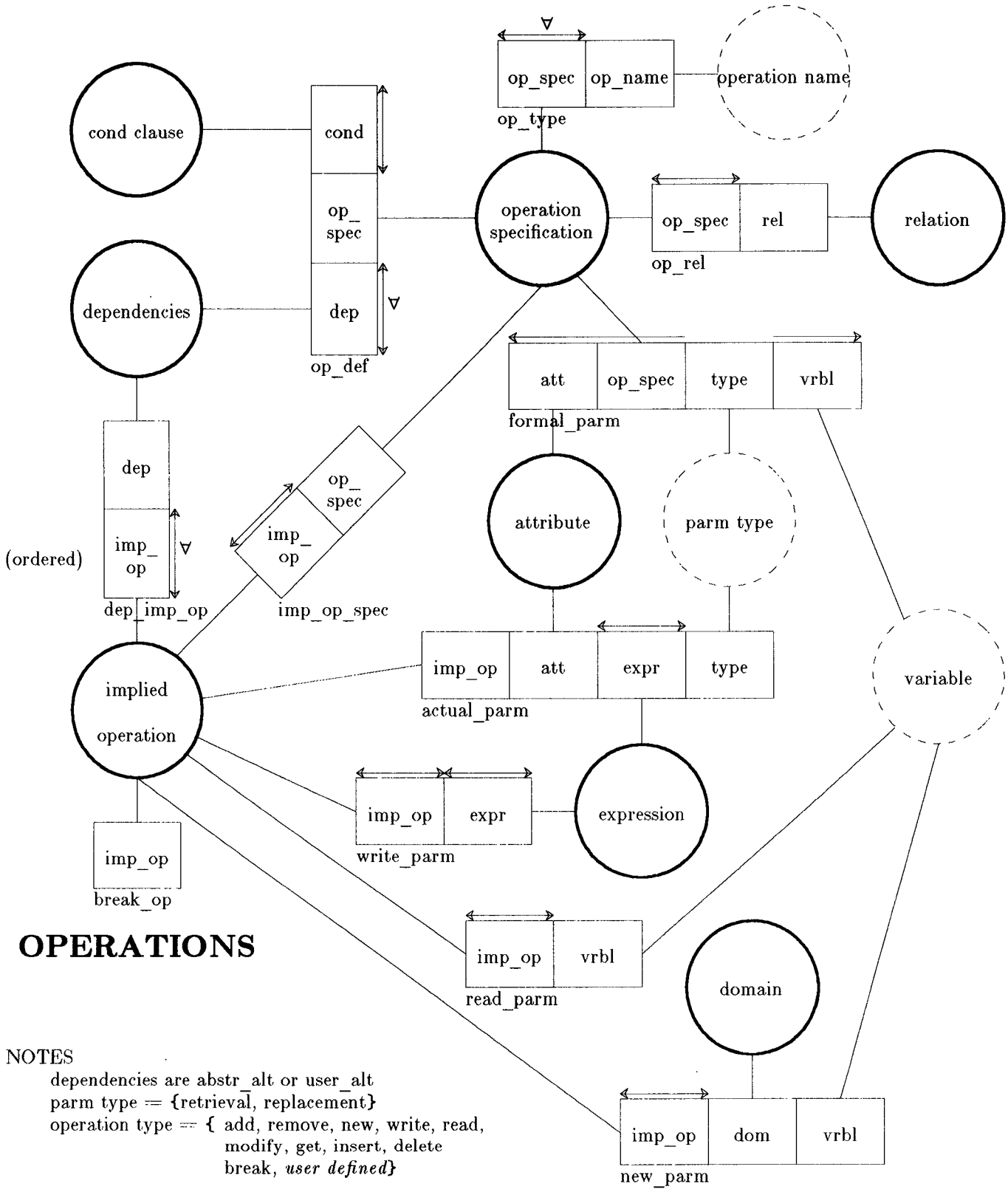
Universal Quantifier, $\forall$, over an attribute indicates that a totality constraint exists for that attribute over the attributes corresponding domain.

The Meta–Schema consists of two parts: the conceptual schema and the external schema. The conceptual schema, presented first, contains the base relations of the meta–schema. The external schema, presented last, contains views derived from the meta–schema for interfacing with the compiler. In general the external schema contains views that are joins over the base relations with most of the surrogates projected out, giving a lexical view of the meta schema.

Both the conceptual schema and the external schema are presented in four logically grouped pages, labeled by the logical group. The first logical group consists of the relations/views that comprise the relational model extended to include a strongly typed semantic net of domains. The second logical group consists of the relations/views that comprise the storage of operation definitions specified in Update Dependencies. The third and fourth logical group define the storage of conditions and expressions, respectively, that can occur in the operation definitions.



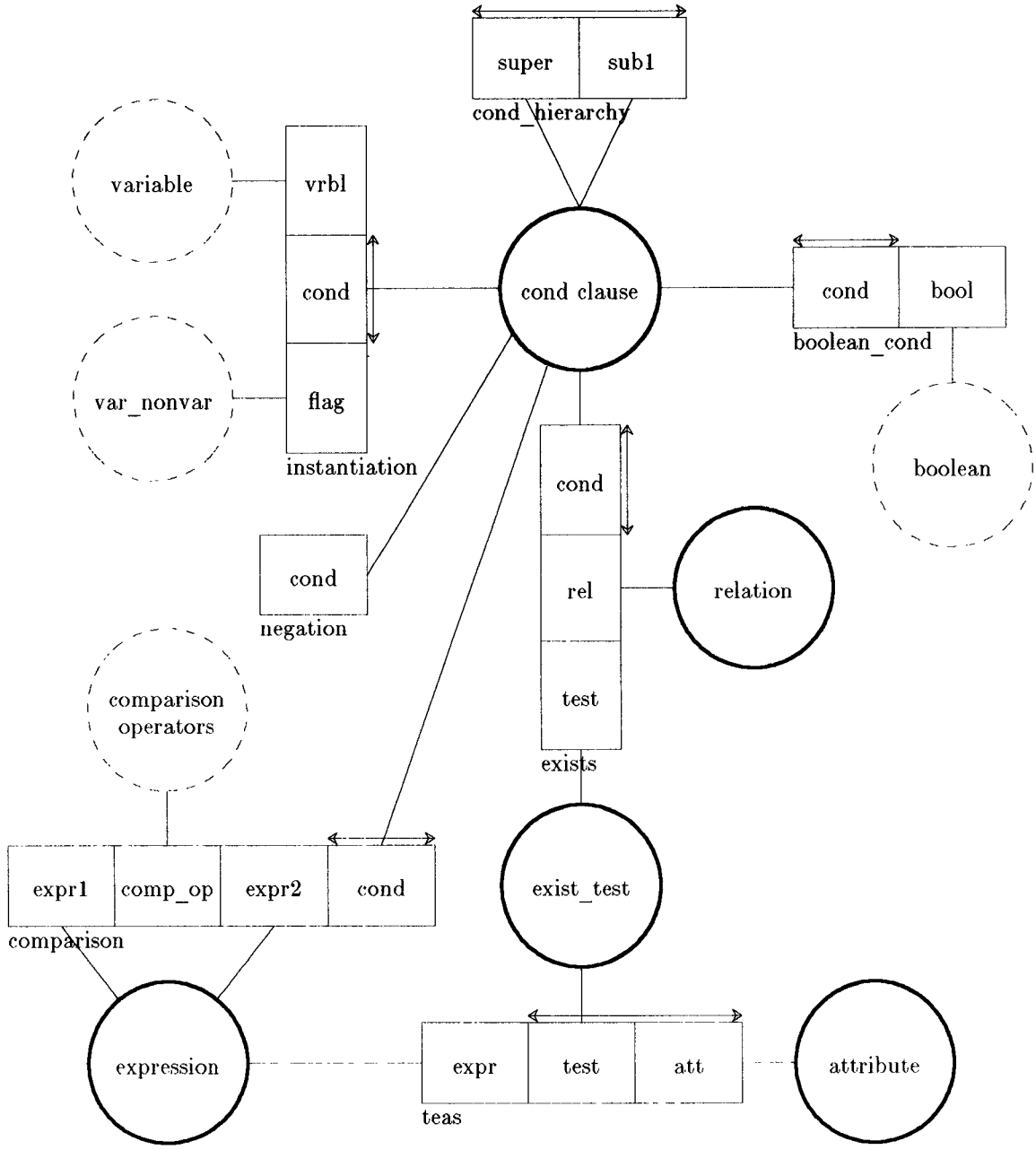
RELATIONS, DOMAINS, ATTRIBUTES and VIEWS



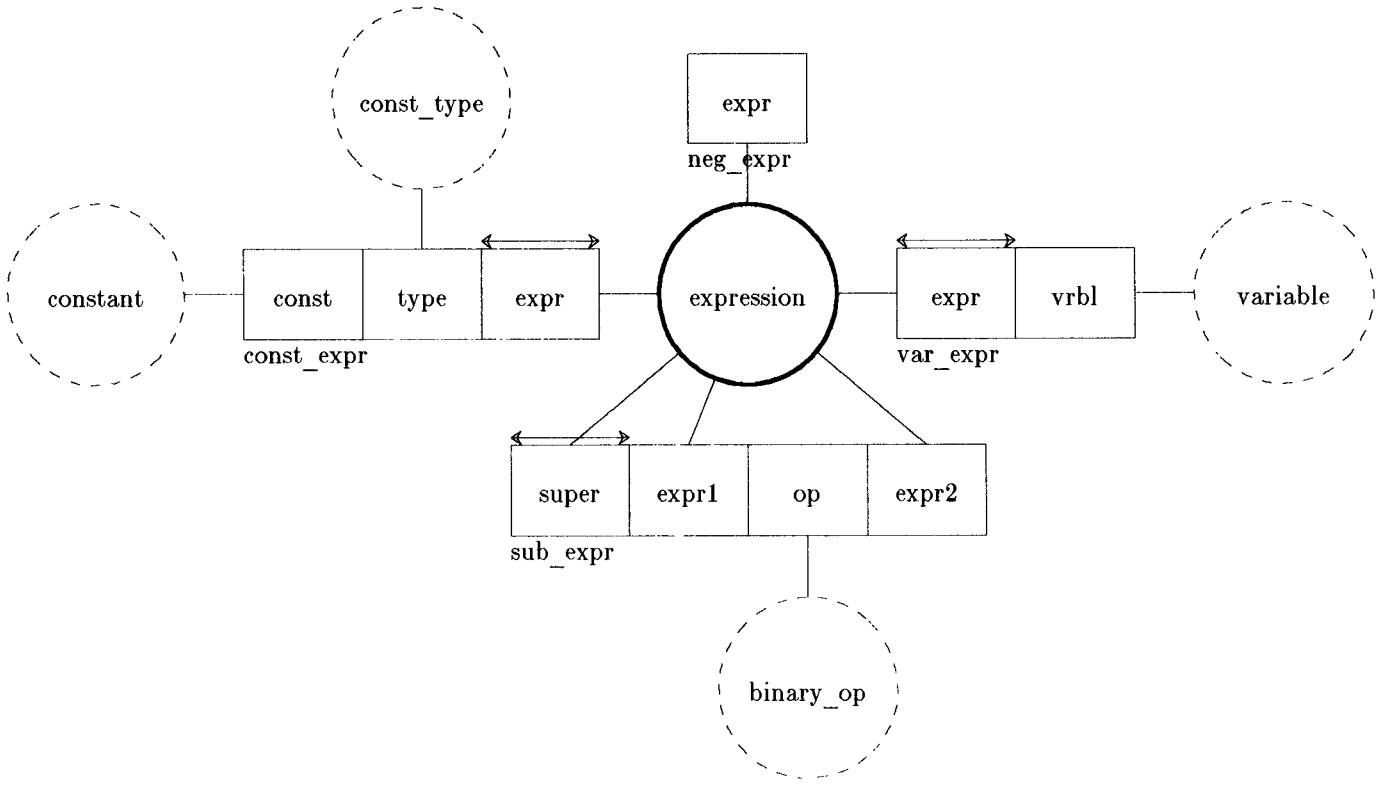
OPERATIONS

NOTES

dependencies are abstr_alt or user_alt
parm type = {retrieval, replacement}
operation type = { add, remove, new, write, read,
modify, get, insert, delete
break, *user defined*}



Condition Clause



Expressions

Appendix C – Semantic Actions

This section contains the semantic actions that accompany the syntax of the specification language. It is comprised of two parts: the annotated syntax, and the meta–schema definitions and operations.

As can be expected, this appendix is very large (71 pages) and is not included here. It can be obtained from the authors upon request.