Symbolic Execution of the Gist Specification Language

Donald Cohen

Information Sciences Institute
4676 Admiralty Way, Marina del Rey, Ca. 90291[1]

Abstract: Symbolic execution can help clarify the behavior implied by a program specification without implementing that specification, and can thereby assist the difficult process of developing a correct specification. However, symbolic execution of specifications poses problems that do not arise in symbolic execution of ordinary programming languages. We describe a symbolic evaluator, named KOKO[2], for the Gist specification language, and show how it copes with such high-level constructs as nondeterminism, constraints, and reference by description

1. Introduction

Current research at ISI approaches software development in two steps. First a user translates his informal requirements into a formal specification. The specification language. Gist [Balzer 81]. is significantly more powerful and expressive than a programming language, allowing the user to concentrate on specifying *what* is required rather than *how* it is to be accomplished. The second step is the implementation of the specification with the help of a transformation system. A sound transformation system guarantees a correct implementation of the specification To support this specification-based paradigm we are developing tools to help a user create a specification, to explain its behavior to the user, and to allow a user to guide the implementation process at an appropriate level [London & Feather 82] [Fickas 82]. Two tools especially relevant to this work are the Gist to English paraphraser [Swartout 82] and the Trace explainer [Swartout 83].

An important part of developing a specification is validation, i.e.. trying to be sure that what is specified is what is really desired. Specifications, like programs, may contain widely separated parts that interact in non-obvious ways. Finding such interactions is the job of symbolic execution. Symbolic execution derives information about the specified behaviors and tries to integrate it into a coherent description. This can help the user find errors in the specification by revealing unexpected consequences or increase his confidence by deducing consequences he desires.

2. Overview of Gist - the Problem

Gist was designed to allow people to specify behaviors in a natural way. It models the state of the world as a relational database consisting of a set of objects and relations among them. The world may change by the addition or deletion of relations among objects, the creation of new objects, or the destruction of existing objects (which implies the deletion of any relationships in which they participate). A specification defines a set of allowable behaviors, i.e., sequences of world states. Gist allows reference to objects by description, e.g., a box that contains a big red ball, where Contains, Big, and Red are relations.

Gist is inherently nondeterministic. The descriptive reference above can refer to any box containing a big red ball. (Nondeterministic control constructs are also supported.) However, the set of allowable behaviors may be pruned by constraints. Any behavior which violates a constraint is excluded from the set of permissible behaviors. For example, a constraint that every yellow box be empty after a big red ball is put into a box restricts which box the ball can be put into. Constraints may be regarded as providing arbitrary look-ahead, in that all nondeterministic choices are constrained to the subset which do not eventually force the violation of a constraint.

In addition, Gist provides control constructs such as conditional statements, procedure calls, demons, parallelism and others that are less common.

It should be clear at this point that symbolic execution as the term applies to implementation languages [Clarke 81] is out of the question. Massive (even unbounded) nondeterminism makes the enumeration of execution paths infeasible. Reference by description poses severe problems of aliasing. The way the world changes is more complex than the familiar semantics of assignment statements, and constraints seem to have no analog at all.

3. Example of Symbolic Execution

Following is a very small specification, translated from Gist into English by the Gist paraphraser [Swartout 82]. (This has been edited because the original translation raises issues that we don't want to address here).

There are sexes and persons. Male and female are the only sexes. Each person has one gender which is a sex. Each person may have a spouse which is a person.

To marry a person p2 to a person p1, the following happen atomically (simultaneously).

7. *The person p2 becomes the spouse of the person p1.*
2. *The person p1 becomes the spouse of the person p2.*

It is always » squired that for all persons y and x, if the spouse of x is y then the spouse of y must be x.

It is always prohibited that there exists a person where the person's gender is the gender of the person's spouse.

The results of symbolically executing the action "marry" are paraphrased below by hand. (The Trace explainer is currently being extended to explain such results in English.)

Since spouses cannot have the same gender and every person has a gender, no person can ever be his own spouse.

This research is supported by the Air Force Systems Command, Rome Air Development Center under contract No F30602 B1 K 0056. and by the Defense Advanced Research Projects Agency under contract No MDA903 81 C 0335 Views and conclusions contained in this report are the authors and should not be interpreted as representing the official opinion or policy of RADC, DARPA. the U.S. Government, or any person or agency connected with them

[2] Koko was the Lord High Executioner in Gilbert and Sullivan's Mikado [Sullivan 65] He never actually executed anyone, but he did manage to describe in detail what would have happened if he had.

Suppose two persons, p1 and p2, get married. Afterwards, p1 is the spouse of p2 and p2 is the spouse of p1. Since spouses of the same gender are prohibited, p1 must not have the same gender as p2. Also, p1 is distinct from p2. Before the marriage p1 must have no spouse other than p2 and p2 must have no spouse other than p1

This example confirms some of our expectations, e.g., that a person cannot marry himself. On the other hand, one might have expected that the two people originally have no spouses, KOKO shows that this is not necessarily the case. Since Gist allows the insertion of relations that already hold, there is nothing in the specification to prevent the marriage of people who are already married -■ provided that they are already married to each other! If this is not what the specifer intended, then symbolic execution has revealed a bug in the specification.

4. Overview of the Solution

Our approach to symbolic execution regards a specification as a large set of domain axioms, expressed in a first order temporal logic with typed variables. The axioms define the set of acceptable behaviors, i.e.. the specified behaviors correspond to the models of the set of axioms. Symbolic execution is a process of forward inference, computing consequences of these axioms Notice that a specification need not determine the truth or falsehood of every relation, i.e., a relation may be true in some behaviors and false in others.

This approach factors symbolic execution into two processes. First, each statement in the specification is translated into axioms about successive world states. Second, these axioms are used to derive certain interesting consequences, e.g.. hidden interactions among different parts of the specification. The success of this approach depends in large part upon the ability of the forward inference engine to find interesting consequences and avoid uninteresting ones However, the control of forward inference is outside the scope of this paper. The rest of the paper describes how various Gist constructs are treated as axioms. We start with primitive constructs and then show how compound constructs are handled in terms of their components.

5. Constraints

Constraints are the easiest Gist construct to handle, in that they are already in the form of axioms. For example, the constraint that the spouse relation be symmetric is expressed as

$$\forall s_{state}\, x_{person}\, y_{person}\ (\text{Spouse}(x,y) \supset \text{Spouse}(y,x))\ \text{in}\ s$$

Actually, in the current implementation, facts about different states are stored separately: more on this later.

6. Descriptive Reference

Part of the meaning of a Gist statement like the constraint "require Contains(a box. a ball)" is that there must be referents of the object descriptions. KOKO creates a typed "symbolic instance" for each such description. If we call these symbolic instances boxI and ball2, symbolic execution simply proceeds by adding the axiom Contains(box1,ball2). The interpretation in which this makes sense is that boxI and ball2 are not actually objects in the world, but rather names of objects. The distinction is that several names can refer to the same object. Thus we do not preclude the possibility that ball2 is actually the same object as some other ball that was referred to earlier.

Descriptive reference is merely a constrained form of nondeterministic reference, e.g., requiring a box to contain a red ball is modelled by adding the axioms Contains(box1,ball2) and Red(ball2).

One kind of consequence KOKO considers interesting is that two descriptions must (or cannot) refer to the same object. Specifications often contain constraints that imply the identity or non-identity of such descriptions. The most common such constraint requires that a relation be a single-valued function of one of the arguments, e.g., Gender. Another common constraint specifies that a relation describes an optional attribute, e.g.. Spouse.

The consequences of uniqueness constraints are found by forward inference. For instance, from Spouse(pl,p2) and Spouse(pl,p3) KOKO deduces p2 = p3. Conversely, from Spouse(pl,p2) and ~Spouse(pl,p3) it deduces p2*p3. KOKO also uses uniqueness constraints to find consequences of facts with universally quantified variables or several arguments to compare, e.g., from Gender(pl,sexI) and Gender(p2,sex2) it deduces p1 ≠ p2⊃sex1 = sex2.

7. Primitives that Change the World

The most direct effects of primitive changes are easy to axiomatize, e.g.. in the state after "insert Spouse(pl,p2)" it is required that Spouse(p1,p2) However, such constraints cannot completely capture the effect of a change. In particular, first order predicate calculus cannot represent the notion that the before and after states are the same except for the effects of the change.

This notion is captured by predicate transformers [Dijkstra 76]. KOKO stores each state explicitly along with the set of tacts known to be true in that state. Facts are propagated between neighboring states. Notice that propagating a constraint backward in time allows KOKO to identify its implications for earlier choices. We use Pre(S.F) to denote the consequences KOKO derives about the state preceding execution of the statement S given that the tact F holds afterward Similarly. Post(S.F) denotes the consequences about the state resulting from S given that F was true beforehand. For readability, we use the notation "F1 before S => afterwards F2" for "Post(S,FI) = F2" and "F1 after S => beforehand F2" for "Pre(S.FI) = F2".

The computation of pre- and post-conditions'is considerably simplified by the following considerations. For any executable statement S and any propositions P and O

$$\text{Pre}(S, P \wedge Q) \equiv \text{Pre}(S,P) \wedge \text{Pre}(S,Q)$$
$$\text{Pre}(S, P \vee Q) \equiv \text{Pre}(S,P) \vee \text{Pre}(S,Q)$$
$$\text{Post}(S, P \wedge Q) \equiv \text{Post}(S,P) \wedge \text{Post}(S,Q)$$
$$\text{Post}(S, P \vee Q) \equiv \text{Post}(S,P) \vee \text{Post}(S,Q)$$

Quantifiers are eliminated by skolemization. This reduces the problem of computing pre and post-conditions of general propositions to the special case of literals i.e., positive or negative instances of relations with constants, universally quantified variables, and function applications as arguments. (In the rest of this paper, "x" and "y" are universally quantified variables, "f" and "g" are functions, and other unquantified symbols are constants.) The rest of this section describes how this is done for different kinds of primitives.

7.1. Changing a Relation

In Gist, the insert and delete statements add and remove relations. The relation is named explicitly, but the objects may be named by description (and thus nondeterministically), e.g.. "insert Red(a ball)". Only one instance of a single relation is changed by each such statement; "insert Red(ball) V Green(balM)" has no meaning. An insertion results in a new state containing the inserted fact. Deletion is treated as insertion of a negated fact.

The problem of computing pre- and post-conditions of other facts with respect to insertion and deletion is simplified by the following considerations. Gist is a first order language, i.e., there are no variables ranging over relations. Thus any literal whose

relation differs from the one being inserted or deleted is unaffected. (Gist supports "derived" relations, whose values are changed by changing other relations, but the handling of these is outside the scope of the present paper.) Also, if a positive literal is true before an insertion or a negative literal is true before a deletion, it will still be true afterward:

Red(ball1) before insert Red(ball2) = >afterwards Red(ball1)

Finally, if a positive literal is true after a deletion or a negative literal is true after an insertion, it must have been true before:

~Red(ball1) after insert Red(ball2) = >beforehand ~Red(ball1)

The only cases in which insertion or deletion changes an existing fact are those in which a literal true beforehand is changed by the insertion or deletion, or a literal that is true afterward is made true by the insertion or deletion. This happens just when the arguments of the fact all refer to the same objects as the corresponding arguments of the relation being changed:

P(a,x,f(x))) before delete P(b,c,d) = >
afterwards P(a,x,f(x))∨(a = b∧x = c∧f(x) = d)

Recall that all of the variables are universally quantified, so after deleting P(b,c,d), P(a,x,f(x)) is still true for all x with the possible exception of c, and that is only an exception if a = b and f(c) = d.[3]

7.2. Changing the Type of an Object

Types in Gist may be thought of as unary relations. Thus "p is a person" corresponds to Person(p). An object's type can be changed by a Gist reclassification statement, e.g., "Pinocchio becomes a person", which corresponds to "insert Person(Pinocchio)".

Quantifiers in Gist always range over objects of a particular type. Therefore a universal statement may have exceptions in neighboring states where more objects have the specified type, e.g.,

$\forall x_{person} \exists y_{person}$ Mother(y,x) before
Pinocchio becomes a person = > afterwards
$\forall x_{person} \exists y_{person}$ (Mother(y,x) ∨ x = Pinocchio)

This is exactly the effect that arises from treating types as unary relations: $\forall x_{person} \exists y_{person}$ Mother(y,x) means

∀x∃y(Person(x)⊃(Person(y)∧Mother(y,x))),

where x and y are now untyped variables. KOKO would skolemize this to (Person(x)⊃(Person(f(x))∧Mother(f(x),x))),[4] translate "⊃" in terms of "∨" and "~", and apply the rules for computing a post-condition:

~Person(x)∨(Person(f(x))∧Mother(f(x),x)) before
insert Person(Pinocchio) = > afterwards
~Person(x)∨x = Pinocchio∨(Person(f(x))∧Mother(f(x),x))

This is equivalent to the post-condition above.

7.3. Creating and Destroying Objects

In Gist, objects are always created with a type. Creating an object is like inserting a type relation except that (1) the created object is different from any object that ever existed before and (2) this object is in only those relations inserted since its creation. (Objects in the initial state may be in arbitrary relationships as long as no constraints are violated.) Similarly, destroying an object is like deleting a type relation except that (1′) the destroyed object differs from any object that ever exists after the destruction, and (2′) destruction deletes all relations in which the destroyed object participated.[5]

These properties of creation and destruction cannot be expressed in first order predicate logic; instead, they are embodied in the inference engine and the predicate transformers. For example, in simplifying an equality KOKO checks to see whether one object existed before the other was created. The pre- and post-conditions of creation and destruction combine the effects of reclassification with the requirement that non-extant objects cannot participate in relations.

Notice that the create and destroy statements are almost symmetric in the sense that each viewed backward in time looks like the other. The only difference is that a destroy statement deletes all of the relations involving the destroyed object, whereas the create statement is not empowered to insert arbitrary relations involving the created object.

The following table summarizes the conditions under which creations and destructions invalidate literals. We use P(x,a) as a representative literal, where "a" is a constant and "x" is a universal variable. The occurrences of "a = c" represent the condition that the created or destroyed object (c) is one of the parameters of the relation (we exclude variables since in one case they simplify out and in the other case they are included by the other condition), e.g.,

P(x,a,f(g(x))) before destroy c = > afterwards
P(x,a,f(g(x)))∨c = a∨c = f(g(x))

The occurrences of "x = c" represent the condition that the created or destroyed object is (instantiated by) any of the (universal) variables in the literal, e.g.,

P(f(x,y)) before create c = > afterward P(f(x,y))∨x = c∨y = c

Additional detail is contained in notes [i] - [vi] below.

Pre- and post-conditions for create and destroy

P(x,a) before create c = > afterwards P(x,a)∨x = c [i]
P(x,a) after destroy c = > beforehand P(x,a)∨x = c [i]
~P(x,a) before create c = > afterwards ~P(x,a) [ii,v]
~P(x,a) after destroy c = > beforehand ~P(x,a)∨a = c∨x = c [v]
P(x,a) before destroy c = > afterwards P(x,a)∨a = c [iii,vi]
P(x,a) after create c = > beforehand P(x,a) [iii,vi]
~P(x,a) before destroy c = > afterwards ~P(x,a)
~P(x,a) after create c = > beforehand ~P(x,a)

[i] a = c is impossible unless the prior state was devoid of objects of the same type as x, i.e., the quantifier was vacuous. In this case P(x,a)∨x = c still holds if a = c.

[ii] If creations were allowed to insert relations containing the new object, this entry would be "~P(x,a)∨a = c∨x = c".

[iii] x = c reduces to false here since the quantified x only refers to objects that exist.

[iv] Since creation does not insert relations, this case could only arise if insertions were done at the same time as the creation. See section 8.2. If creations were allowed to insert relations of the new object, this would be "P(x,a)∨a = c". The x = c reduces to false. See [iii] above.

[v] When P is the equality relation, x≠a, the result is x≠a∨x = c

[vi] When P is the equality relation, x = a, the result is x = a

8. Compound Statements

In order to save space we describe only a few problematical constructs. It should be obvious how sequences and procedure calls can be handled. Conditionals are not hard, given a way to

[3] Readers familiar with unification will notice both similarities and differences. For example, P(f(a)) would match with P(f(b)) giving the "substitution" (f(a) = f(b)), i.e., it is not necessary that a = b, just that f(a) = f(b). Also, it is perfectly acceptable to unify x with f(x). In a sense we have a generalized version of unification which can be used for theorem proving, e.g., P(a,b)∨Q is resolved with ~P(c,d)∨R to give a≠c∨b≠d∨Q∨R.

[4] To make skolemized axioms well-defined, we treat skolem functions as immutable and defined over all objects that ever exist, but do not constrain the value of the skolem function on objects outside the original type.

[5] The equality relation can relate non-extant objects and is considered to be immutable. The table below is suitably altered for this case

represent the truth of the branch condition as of the branching state. It should be mentioned that the branches are combined into a common state after a conditional, i.e., rather than producing a tree of behaviors, KOKO describes the state after the conditional in terms of which branch was taken.

8.1. Loops

We distinguish between "simple" loops, which can currently be handled and "non-simple" loops which cannot. Simple loops are those in which the iterations are independent of each other, i.e.. the same thing is done to each of a set of objects, as in "move all old files off line"

Most loops in implementations are not simple, e.g.- "for each file, if age(file)>age(oldestfile) set oldestfile to file' However, these loops tend not to appear in Gist specifications. They are replaced by descriptive reference, e.g.,

"a file 1 such that Vfile2 age(file1)>age(file2)"

Simple loops are symbolically executed for the entire set at once. Basically, the loop variables turn into universally quantified variables in the facts that are inserted. After "if file1 is old move it offline" we know old(file1)⊃offline(file1), whereas after "for all files F, if F is old move it offline" we know $\forall x_{file}$ old(x)Doffline(x). All of the symbolic instances that are generated in a loop are skolem functions of the loop variables. In general the computation of pre- and postconditions introduces existential quantifiers, but is otherwise similar to the versions described above, e.g..

~P(a,b) before insert P(c,d) => afterwards
~P(a,b)∨(c = a∧d = b). whereas
~P(a,b) before insert P(x,f(x)) => afterwards
~P(a,b)∨(∃x x = a∧f(x) = b))

8.2. Atomic Statements

The Gist "atomic" construct combines the effects of several constituent statements into a single state transition. An example is the marriage action that simultaneously inserts two spouse relations. It would not have been sufficient to insert one at a time because this would have led to an intermediate state of the world that violated the constraint that the spouse relation be symmetric (Actually, that specification would still have been consistent, but now it would be possible to marry two people only if they were already each others' spouse •• another interesting result of symbolic execution.) Of course, the constituent statements of an atomic must themselves cause no more than one state transition.

The facts that become true because of the statements in the atomic must all be true in the final state, e.g., if an atomic contains both insert P(a) and delete P(b), then a and b must be distinct. This points out a difference between executing two statements atomically and executing them in either order. There is no problem with inserting P(a) and then deleting it. A fact that is propagated through an atomic can be affected by any combination of the statements in the atomic. The pre- or post-condition of a fact with respect to an atomic statement is the disjunction of the pre- or post-conditions of the fact with respect to each constituent statement.

9. Conclusion

We have described a system that characterizes the behaviors permitted by a formal specification containing such constructs as descriptive reference, nondeterminism, and constraints. It translates a specification into a set of axioms and uses forward inference to compute interesting consequences of them. It uses predicate transformers to propagate facts between neighboring states; the computation of pre- and post-conditions in the relational database model has, to the author's knowledge, never been described before.

We have been pleasantly surprised to find that, although many problems that arise are very difficult (or even impossible) to solve in general, the most common and useful cases tend to be the easiest. We have also found that a high level specification can be easier to execute symbolically than a low level program. In retrospect this is not surprising, since the characterization of low level implementations involves a lot of work that could be described as de-compilation.

The decision to represent each state explicitly imposes certain limitations. In particular, arbitrarily long sequences of states cannot be represented. This precludes the description of non-simple loops and certain types of historical reference Historical reference (a special case of descriptive reference) is not yet handled. We also currently do not attempt to handle the arbitrary interleaving and merging of lines of control provided by Gist. We hope to attack these problems, but a great deal can be done without solving them. In particular, KOKO examines the "execution" of one line of control in isolation

KOKO has produced fairly complete descriptions of some small but non-trivial specifications Sample domains include a simplified postal package router, a world of ships and a simplified file system. Of course we expect to increase the coverage of the specification language so that more specifications can be so characterized. We believe that even without solving the difficult problems that remain, KOKO can be extended to characterize the behavior of a large class of interesting specifications.

Acknowledgements: This work was done in the context of a larger effort on the part of the Gist group at ISI. In particular, the specification language and the entire approach to the development of software defined the problem whose solution is presented (in part) here. This paper was greatly improved by the suggestions of Jack Mostow and other members of the group.

References

[Balzer 81] Balzer R . *Design specification validation.* University of Southern California Information Sciences Institute Technical Report. 1981. Published by Rome Air Development Center as RADCTR-81-102

[Clarke 81] Lon A. Clarke, Debra J. Richardson. *Symbolic Evaluation Methods,* University of Massachusetts at Amherst, Technical Report COINS TR81 -8. May 1981.

[Dijkstra 76] Edsger W. Dijkstra, *A Discipline of Programming,* Prentice Hall, 1976.

[Fickas 82] S Fickas. *Automating the Transformational Development of Software.* Ph.D. thesis. University of California at Irvine. 1982.

[London & Feather 82] London. P.E. & Feather. M.S.. "Implementing specification freedoms." *Science of Computer Programming,* (2). 1982. 91-131.

[Sullivan 85] Sir Arthur S. Sullivan & W. S. Gilbert, *The Mikado.* W. A. Pond & Co.. New York. 1885.

[Swartout 82] Bill Swartout. "Gist English Generator." in *Proc. AAAI-82,* pp. 404-409. August 1982.

[Swartout 83] Bill Swartout, The Gist Behavior Explainer. 1983. Submitted to AAAI83-

MANIPULATING DESCRIPTIONS OF PROGRAMS FOR DATABASE ACCESS

P.M.D.Gray and D.S.Moffat

Dept. of Computing Science,

University of Aberdeen, Scotland, U.K..

ABSTRACT

A method is described for manipulating descriptions of programs to access Codasyl Databases to meet a specification given in relational algebra. The method has been implemented as a Prolog program which is compared with the previous Pascal version. The methodology is discussed as an Automatic Programming technique which explores the transformations on a program induced by changes of data structure representation at two levels.

I INTRODUCTION

The problem of generating equivalent programs under changes of data representation is an important one. In the case of list processing, a change of data structure representing sets of objects and their relationships can completely change the program. The same applies to Codasyl databases which are essentially enormous list structures on secondary storage. However because of the variety of redundant pointers it is possible to traverse the same list structure in many different ways. Thus it is not just a question of changing the program but of generating alternative programs whose run-times, because of disc access, may differ by factors of 10 or more.

This paper concerns the manipulation of abstract descriptions of such programs. A query is formulated in a functional language (relational algebra) which specifies the logical relationships between the retrieved data values and the stored data items but does not specify the sequence used to access them (the access path). The aim is to generate a program that produces the desired items efficiently by exploring a variety of alternative program structures, which are the consequence of following different access paths.

A method of doing this has been developed (Bell 1980) and embodied in a system (ASTRID) (Gray 1982) for typing in queries in relational algebra and generating and running programs on Codasyl databases (IDS-II and IDMS). From the user's point of view the benefits are twofold.

1. It gives the user a relational view of the Codasyl database. Thus he is able to think about his retrieval problem in terms of table manipulations using the high level operations of relational algebra instead of having to work at the low level of record access operations following pointers through the database and embedding these operations in Fortran Code.

2. He can write complicated multi-line queries that compute derived data both from records and groups of records (averages, counts etc.) and appear to generate several intermediate tables. The system will endeavour to find an access path that computes the same result without storing these tables, which could be very costly for large databases. The program generated may be quite complicated to write by hand and should be competitive with a trained programmer's code.

The system goes through several stages. First the user types a query in relational algebra which is parsed and checked. Then it is manipulated at two levels. At the top level the query is rewritten still in algebraic form using rewrite rules so as to assist transformations at the next level. The lower level uses a concrete representation of the Codasyl data structure by a traversal (see below). The system reads in a number of stored traversals for each relation. These have each to be manipulated and combined in various ways to satisfy the requirements of the query. Some combinations will represent very slow and inefficient programs and be discarded. However this cannot be done immediately, as a good program for part of the query may later turn out to be second best after modification to fit the remainder. Finally the descriptions are costed according to information on database access times and the selected version is used to generate Fortran code to run against the actual database. The system is oriented towards complex queries accessing thousands of records which can only run in batch producing substantial printout. Thus it is not the run-time for the translator which matters but the complexity of query which it can handle. Currently other systems only handle a very restricted relational view or a rather restricted query language.

The ASTRID system was originally written in Pascal. More recently the two levels of manipulation have been rewritten in Prolog. This paper describes the basic methodology and shows how Prolog is well adapted to this task.

The layout of the paper is as follows. Section {II} describes some transformations which affect the resultant program but are best carried

out on the relational algebra in Prolog. Section {III} describes the basic notion of a traversal and how it is used to represent a piece of program. Section {IV} describes the combination of traversals and how this is used to build descriptions of more complex programs. Section {V} illustrates some of the Prolog used to combine traversals and discusses its advantages and snags in this application. The final section draws conclusions for future work.

A. Relation to Other Work

Burstall and Darlington (1977) describe a system for specifying a program by recursion equations. These can be manipulated and play a role similar to relational algebraic expressions in our system. They discuss a way to rewrite the abstract program given a concrete data representation in terms of a "coding function". However our use of a traversal represents the data in a rather different way. Apart from Tarnlund (1978) few have addressed the problem of efficient access to relations using information about the mode of storage. Tarnlund has studied ways to answer queries efficiently by representing them as theorems to be derived in the first order calculus and looking for efficient derivations where relations are held as a binary tree structure.

II_ RELATIONAL ALGEBRA TRANSFORMATIONS

The user asks his query in relational algebra. We first describe this and then see how the system improves the query by rewriting it.

A. Relational Databases

A relation is a set of tuples each containing values for a fixed set of attributes. Viewed as a table the attribute values are in columns. A relational database usually contains several relations which have attributes in common. The examples used come from a database on World Cup football results. The two relations of interest are shown in Table 1.

Table 1. Relational View of World Cup Database

STADIUM_ALLOCATION

year	group	game	stadium	date
1978	1	1	Buenos Aires	2 Jun
1978	1	2	Mar_del_Plata	2 Jun
...		...		

GROUP_PLACINGS

year	group	team	placing
1978	1	Italy	1
1978	1	Argentina	2
...		...	

B. Relational Algebra

Relations can be treated as tables and new relations derived from them by the operations of relational algebra. The operations used are adapted from Codd. They are selection, projection, join, extend and group_by (Gray 1981). The join operation is a generalised intersection, formed from the cartesian product of two relations by selecting those tuples with matching values for the common attributes. A typical query starts by joining several relations, then selects tuples, then extends and or groups these tuples and finally projects to required columns.

The relational algebra can be rewritten, just like standard algebra, by using rewrite rules in PROLOG. We have 17 such rules with special predicates for handling commutation. A typical transformation would move a projection operation(%) in an expression involving join(*) and selection(;) to ease the join method.

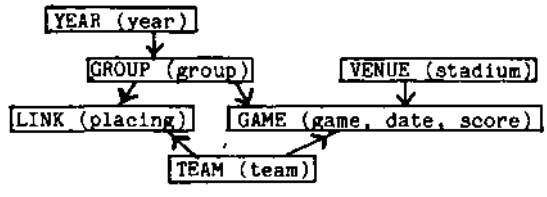
```
(STADIUM_ALLOCATION ; [stadium = "Cordoba''] %year,
 group)*(GROUP_PLACINGS  ;[placing=1]  *year,group)
          becomes
(STADIUM_ALLOCATION  ;[stadium="Cordoba"]  *
 GROUP_PLACINGS  ; [placing = 1 ])  %year,group
```

III TRAVERSALS of CODASYL DATABASE STRUCTURES

Although the user thinks of relations just as tables, they are actually complicated doubly-linked list structures. At the second level of transformation we need to represent possible paths through these structures by traversals in order to search for an efficient one. Thus we first explain the Codasyl "set" relationship used to link different records. We then see how a number of alternative "base traversals" can be defined for each relation and held on file.

A Codasyl database consists of sets of records of the same type which are linked by pointers to other records in the set and to a common owner record which uniquely identifies an instance of a given set type. Figure 1 shows the linkages between records in the World Cup database.

Figure 1. Bachmann diagram of World Cup Database



A. Traversals

We can now define a traversal of a relation more precisely. It is a description of a piece of code which realises the tuples of the relation one

at a time by accessing the records in some sequence following the set pointers and modifying the values as necessary. Thus it is a generator for a relation. Corresponding to each relation stored in the database (e.g. GROUP_PLACINGS) we hold on file one or more base traversals. Each one is essentially a description of a piece of code with a number of nested loops.

We have a notation for traversals as follows. Internally it is represented by a Prolog list structure. There are three obvious base traversals of STADIUM-ALLOCATION and two for GROUP_PLACINGS. Each {SA} traversal visits the same number of GAME records, generating one tuple for each.

```
S(YEAR) -> D(GROUP) -> D(GAME) -> U(VENUE)   {SA1}
V(VENUE) -> D(GAME) -> U(GROUP) -> U(YEAR)   {SA2}
B(GROUP) -> U(YEAR) -> D(GAME) -> U(VENUE)   {SA3}
S(YEAR) -> D(GROUP) -> D(LINK) -> U(TEAM)    {GP1}
B(GROUP) -> U(YEAR) -> D(LINK) -> U(TEAM)    {GP2}
```

Here S means a singular set access to visit-all records of a given type (there is only one set owning all year records), D mean go down to visit all member records belonging to the given owner using the appropriate set type (if this is ambiguous it is specified) and U means go up to visit the owner of a given record, V means direct access to the record containing a value (usually given by selection). B means vi it every record of that type in the database, In an Algol-like syntax we can represent the corres ponding code for SA1 as :-

```
for each YEAR record do
  for each GROUP record owned by YEAR do
    for each GAME record owned by GROUP do
      for the VENUE owner of GAME do
        print YEAR.year, GROUP.group, GAME.game,
              VENUE.stadium, GAME.date.
```

Thus each arrow in a traversal represents an inner level of nested code. Note that the record generations such as D(GAME) in SA3 must follow those such as B(GROUP), which generates the owner for GAME, but they need not be consecutive.

IV COMBINATION & MODIFICATION of TRAVERSALS

Corresponding to every algebraic operation on a given relation there is a modification to Its traversal which produces a derived traversal, which is a generator for the new relation. Thus the method is complete. This derived traversal can then be modified by the next operation and so on. For example a selection can be done by inserting "if (year=1978) then" just after "for each YEAR record do". The resulting traversal depends somewhat on the order of application of operations specified by the user. However many of these are commutative and the order of others can be improved by top level rewriting.

A. Combination by JOIN

Since Join is based on a cartesian product it can be formed by a nested for loop with one iteration for each record type involved. This is very similar to a traversal structure and it turns out that the traversal representing the join can often be formed just by concatenating parts of the separate traversals {Bell 1980, Gray 1981}. The selections for matching are then performed automatically by the fact that a Codasyl owner record will in many cases be linked to just those records whose values would have been selected by the join operator! Let us consider examples of this using

RES:= STADIUM-ALLOCATION joined_to GROUP-PLACINGS

If we use SA1 and GP1 then these both have "common start" section.

S(YEAR) -> D(GROUP)

which generates the common attributes in the two cases. If we concatenate the traversals keeping one copy of the common start we get

```
S(YEAR) -> D(GROUP) -> D(GAME) -> U(VENUE) ->
                       D(LINK) -> U(TEAM)
```

we can also get in the other order :-

```
S(YEAR) -> D(GROUP) -> D(LINK) -> U(TEAM) ->
                       D(GAME) -> U(VENUE)
```

Both traversals correspond to nested loop code which will produce the desired tuples though in a different sequence. Which is best depends on subsequent selections. If a selection on "placing=1" is made after "D(LINK)" then the second method is best as it visits fewer records.

One can also join traversals where the head of one traversal matches the tail or middle of the second. We can do this with the alternative traversals SA2 & GP2 giving :-

```
V(VENUE) -> D(GAME) -> U(GROUP) -> U(YEAR) ->
                       D(LINK) -> U(TEAM)
```

We notice here that a B(GROUP) since it visits all records can match a U(GROUP) which visits only certain records because join has the properties of an intersection.

The second traversal (using SA2,GP2) would be preferred if a subsequent selection were made on stadium as it could use V(VENUE) efficiently. General conditions for choosing an optimum are discussed in (Esslemont & Gray 1982).

1 OVERVIEW of the JOIN ALGORITHM in PROLOG

The basic method is given in Figure 2. It starts by reading in a number of traversals for each relation and holds them as unit clauses trav(X). The term X contains a record generation list giving the sequence of record and set accesses, which we have symbolised. The procedure join trav (see below) then picks the first clause

for each relation and tries to find an overlap in accordance with the conditions given in (Bell 1980). Prod overlap is called twice with the record generation lists reversed in order to try the two cases of common start and likewise for head to tail (IV.A). If this is successful the result traversal is asserted. A 'fail' clause then causes backtracking and another pair of traversal clauses is chosen thus trying all combinations of the operand traversals. The 'fail' also has the effect of reclaiming much-needed space once the traversal is safely asserted. If all attempts fail an operation node to join by sort-merge is inserted.

It is possible for a traversal to pass through two instances of the same record type. In order to distinguish which instance is being used for accessing subsequent record types it is necessary to assign a unique number to each record generation element in the traversal. Correspondences are established by clauses of the form equiv_curr(X.Y).

Figure 2. PROLOG Version of Traversal Join Method

```
join_trav(Rel1,Rel2,Rel3) :-
  common_columns(Rel1,Rel2,ComCol,NumComCol),
  trav(Rel1,_,_,nds_list(Nds1),recg_list(Rg1)),
  trav(Rel2,_,_,nds_list(Nds2),recg_list(Rg2)),
  exists_nondup_list(ComCol,Nds1,Nds2,Rg1,Rg2),
  (retractall(equiv_curr(_)),
  prod_overlap(Rg1,Rg2,Rg3,Rg4,NumComCol) ;
  retractall(equiv_curr(_)),
  prod_overlap(Rg2,Rg1,Rg3,Rg4,NumComCol) ),
  assert_trav(Rel3,Rg3,Rg4),
  fail.
join_trav(_,_,_).
```

A. Effect of Joining Modified Traversals

Traversals which have been modified by selection, extension, projection or group-by will have elements in their record generation lists to indicate these operations(operation nodes). Such traversals are joined as before but with all operation nodes being copied directly into the result traversal.

B. Comparison *of* Pascal and Prolog Versions

The Pascal version takes several thousand lines whereas Prolog needs several hundred and is much easier to read and modify. Pascal is a very much wordier language for list processing. Also one has to write multiple versions of many functions such as "member" because the type of list argument must be known at compile time. Further the use of Prolog Definite Clause Grammars saves pages of recursive Pascal procedures to parse base traversals etc.. Finally because Pascal has no backtracking facilities it has to keep returning sets of alternative combined traversals and currently runs out of list space on large queries. The Prolog version can handle these because it reclaims space following fail.

VI CONCLUSIONS

Although the direct use of Codasyl databases for storage of facts is unlikely in A.I. the general problem of generating programs that traverse and manipulate list structures is important and the techniques described could have other applications. The methodology used is :-

1. Arrange that the specification of the result to be computed by the generated program is given in functional form such as relational algebra but not in procedural form with loops and assignment. This is easier for the user to think about and also does not commit him to an unsuitable representation. It allows easier overall program transformation; in particular some transformations are easier in the functional form than the traversal form.

2. Prolog is particularly suitable for this work because of its good list-matching and back-tracking facilities. The use of "assert and fail" was necessary, but given this it out-performs Pascal by running larger problems in the PDP 11 address space in similar time.

ACKNOWLEDGEMENTS

The rewrite rules described in section II were developed by T.N. Scott (now at SCICON, London). Ben du Boulay gave us many valuable comments during the preparation of this paper. The generous assistance of the U.K. SERC is also acknowledged.

REFERENCES

[1] Bell R., "Automatic Generation of Programs for Retrieving Information from CODASYL Data Bases", PhD Thesis, Aberdeen University,1980.

[2] Burstall R.M. & Darlington J. "A Transformation System for Developing Recursive Programs", JACM, (1977), pp 44-67.

[3] Esslemont P.E. & Gray P.M.D. "The Performance of a Relational Interface to a Codasyl Database" in Proc. BNC0D-2, ed. S.M.Deen and P.H. Hammersley, Bristol 1982.

[4] Gray, P.M.D. "The GR0UP_BY Operation in Relational Algebra", in "Databases (Proc. BNC0D-2)" ed. S.M. Deen & P. Hammersley (1981),pp. 84-98.

[5] Gray, P.M.D. "Use of Automatic Programming and Simulation to Facilitate Operations on Codasyl Databases" in "State of the Art Report DATABASE", Series 9 No.8, ed. M.P.Atkinson, Pergamon Infotech (Jan 1982),pp 346-369.

[6] Tarnlund S-A, "An Axiomatic Data Base Theory" in "Logic and Data Bases", ed. Gallaire & Minker (1978), pp. 259-289.