Compiling a Declarative High-Level Language for Semantic Integrity Constraints

Suzanne M. Embury and Peter M.D. Gray
University of Aberdeen
Department of Computing Science, King's College,
Aberdeen, Scotland, United Kingdom, AB2 9UE.
Tel: +44 1224 272293, Fax: +44 1224 273422
URL: http://www.csd.abdn.ac.uk/~pfdm
E-Mail: {sme|pgray}@csd.abdn.ac.uk

Abstract

The provision of more intelligent support for complex database applications is becoming an important research topic, and declaratively stated integrity constraints are increasingly being seen as a potential source of semantic information for such advanced DBMS's. However, despite much research effort, the problem of efficiently compiling and checking a wide range of complex constraints has not yet been solved. And until it is solved, few database designers will be willing to sacrifice the performance of the every day database usage, in order to provide a complete semantic modelling of their application domain.

In this paper, we describe the implementation of a constraint checking architecture and compiler, which have been designed to combine an expressive constraint language with efficient run-time maintenance. While the current implementation does not yet achieve high efficiency for all constraints, it has been designed with extensibility in mind and forms a solid platform from which to investigate further techniques for improving the handling of a wider range of constraints. The system is currently being used to describe and enforce the semantics of a complex scientific application domain, namely three-dimensional protein structure.

Keywords

Integrity constraints, intelligent databases, automatic code generation, scientific databases, functional databases.

1 INTRODUCTION

Capturing the semantics of database integrity constraints so that they can be enforced centrally by the database management system has long been a goal of database research.

However, there are fundamental problems in implementing constraints by hand-written code encapsulated in update methods. Firstly, it becomes very hard to maintain this code consistently across different methods, especially if different programmers rewrite the methods. Secondly, it is difficult to ensure that all the update methods which must include the constraint have been altered to check it, particularly for more complex constraints. Finally, there is no metadata available about which constraints are being maintained and which can be made available to the user or to other parts of the DBMS — just lots of procedural code spread throughout the update methods.

The solution to this problem is to allow database designers to specify their constraints in a declarative language, and to provide a compiler which generates the procedural constraint checks automatically from the declarative description. However, this approach also has other advantages, the most significant of which is the availability of the declarative constraints as a storehouse of semantic information about the application domain in hand. This information is available not only for use by the users of the database, but also by the DBMS itself, opening up a wide range of possibilities for providing more "intelligent" user support. It allows, for example, the possibility of constraint analysis to aid the database designer (Urban 1992), semantic query optimisation (King 1984), safe transaction synthesis (Qian 1993), automatic transaction repair (Fraternali 1993), declarative support for complex database updates (Embury 1994) and the incorporation of user preferences into queries (Gaasterland 1994).

The main disadvantage of a declarative formulation of constraints, however, is the difficulty of achieving an efficient compilation over a wide range of constraints which does not impose an unacceptable overhead on ordinary database usage. This one factor has probably been the main obstacle to commercial adoption of declarative constraint languages. Unfortunately, it also means that even in systems which do provide such a facility, database designers will be reluctant to specify the more complex semantic constraints required for the kinds of intelligent database mechanisms described above.

With this in mind, we have designed and implemented an architecture for constraint checking in P/FDM (Gray 1992), a Prolog implementation of Shipman's functional data model (Shipman 1981), which aims to minimise the overhead associated with constraint maintenance. We have also extended the Daplex data manipulation language to allow the specification of a wide range of constraints in a high-level and declarative manner. Our compiler is able to generate highly efficient constraint checking code for an important subset of the constraints which are expressible in the language, although we are able to generate some form of check, albeit an inefficient one, for the entire set of expressible constraints. The intention is that this system will form a basis for future research to identify mechanisms by which the checking of a wider range of constraints can be improved. We are already using and evaluating our current system on a database of three-dimensional protein structure, and example constraints from this application are used throughout this paper to illustrate the techniques under discussion.

The remainder of this paper is organised as follows: Section 2 describes some of the approaches which have been taken to the problem of generating efficient constraint code in the past, and highlights the disadvantages of some of these systems. Section 3 introduces the constraint language extension to Daplex, and Section 4 outlines the main features of our approach to constraint maintenance. Sections 5 and 6 describe the constraint language compiler, and in particular the analysis required to determine the set of update operations which may violate a constraint, and the reduction of the constraint to its more efficient form. Section 7 concludes.

2 APPROACHES TO EFFICIENT COMPILATION OF CONSTRAINTS

The main difficulty with implementing a constraint maintenance subsystem is that of minimising the run-time overhead of having to check that constraints are satisfied. Generating code which checks the entire database for satisfaction of some constraint from its declarative specification is comparatively easy for most systems. In fact, in some systems which are based upon a declarative paradigm themselves, constraints are expressed directly as queries, usually with the implied restriction that they will evaluate to the empty set if the database integrity is not violated (Erwig 1991, Reddi 1993). However, having to check the entire database for validity is not an efficient way to check integrity, especially if the query has to be executed after every update operation.

One of the most significant contributions to this problem, which has since been used as the basis for almost all implementations of integrity constraint subsystems (Bry 1992), is the simplification method proposed by Nicolas (1982). Nicolas realised that, if one can assume that the database state satisfies all the integrity constraints immediately prior to an update operation, then it is only necessary to check those constraints which could be affected by that update. So, the constraint that all students are older than 16, for example:

$$(\forall s,\ a)\ (student(s) \wedge age(s,\ a)) \Rightarrow a > 16$$

cannot be violated by the addition of a new `course` to the database, and so need not be checked in this case*. Neither is it possible for any deletion, even of students or ages, to violate this constraint, and it need therefore be checked only in a very restricted number of cases.

Nicolas also showed that it is sufficient to check a *simplified* form of the full constraint, that tests for violation only within the subset of the data that is affected by the update, and, moreover, that this simplified form could be generated at compile-time by considering each of the updates that might violate the constraint in turn. For example, we know that the creation of a new student, S, with age A could violate the above constraint, and we can use this information to partially evaluate it, so that:

$$(student(S) \wedge age(S,\ A)) \Rightarrow A > 16$$

becomes

$$(True \wedge True) \Rightarrow A > 16$$
$$\equiv A > 16$$

*This property relies on the specification of the constraint being range-restricted (Bry 1992). However, all practical constraint languages exhibit this property and this is not a significant restriction on expressibility.

In other words, the update satisfies the full constraint if the new age value, A, satisfies the simplified constraint check. This technique has the advantage of being conceptually simple and is independent of any particular paradigm or architecture. It was originally proposed for relational databases, with constraints expressed in first order logic, but it has also been successfully applied to logic and deductive databases (Bry 1988, Poulovassilis 1991) and semantic data model databases (Urban 1989, Jagadish 1992, Díaz and Embury 1992, Bassiliades 1993).

Systems differ, of course, as to the exact form of the code generated, and also in the amount of simplification that can be performed, which depends on the flexibility of the underlying architectures. To achieve the maximum degree of simplification, it must first be possible to identify exactly which database events could cause the violation of a given constraint, and which may be ignored. Secondly, the architecture must be capable of modelling the many-to-many relationship between update events and constraints, so that a highly simplified code fragment can be produced *for each such event*.

The inefficiencies caused by the first problem can be seen in the Ode system (Gehani 1991, Jagadish 1992). Ode is based upon a C++ data model, in which a high-level statement of each constraint is compiled into several triggers, attached to the various classes involved in the constraint. The authors describe several optimisations which simplify each trigger according to the context of the class to which it is attached, but due to the lack of semantic knowledge about which methods may update which parts of the system, they are unable to specialise the triggers according to a particular update. Nor can they identify which updates to the class may violate which constraints — all constraints on a given class must be checked after all invocations of a public member function, regardless of the semantics of that function.

The Starburst system (Ceri 1990) provides us with an illustration of how an incomplete modelling of the relationship between events and constraints hampers simplification. In this system, which generates situation-action rules from constraint specifications, a single rule is generated for each constraint, that is triggered by any of the set of events that may violate its integrity. The condition of this rule evaluates the set of all tuples not satisfying the constraint and succeeds if it is non-empty. Since the rule may have been fired by any one of the triggering events, it is not possible to use the details of the actual update to simplify the constraint condition at compile-time.

3 A CONSTRAINT LANGUAGE EXTENSION TO DAPLEX

In our own system, P/FDM, interaction with databases is carried out through the Daplex data definition and manipulation language. Daplex is a functional style language that was designed by Shipman as a "conceptually natural interface" to functional data model databases (Shipman 1981). It is based upon the idea of using declarative descriptions of sets of values to provide the arguments to procedural (side-effecting) methods. In general, a Daplex set is based upon some database class, which is then further restricted by some predicate on the instances of that class. For example, the following fragment of Daplex code:

```
p in protein
```

expressed against the schema of our protein structure database, describes the set of all known proteins, whereas:

```
p in protein such that molecular_weight(p) > 2000
```

describes the set of all large proteins in the database. The "such that" keywords here introduce the predicate, which then acts like a filter on the instances of the protein database class.

Daplex provides a rich set of constructs for describing predicates on database objects. Predicates can involve arithmetic expressions, database navigation (expressed using composed functions), aggregate functions such as count, total and minimum, conditional expressions and tests for subclass membership. The syntax for these language features and other constructs for describing predicates is given in Appendix 2. To extend this language to express constraints over database values, then, it is only necessary to add new syntactic constructs for quantifying values over predicates. In fact, Daplex already contains syntax for six different types of quantifier, which are provided for describing quantified expressions. For example:

```
p in protein such that some c in chains(p) has molwt(c) > 2000
```

This describes the set of all proteins that have at least one constituent chain with a high molecular weight. The basic syntax for quantified expressions is:

```
<quantifier> <var> in <set> has <predicate>
```

and the six quantifiers and their semantics are:

- all or each : the quantified expression is true if the predicate is true for every element of the set,
- some or any : the quantified expression is true if some element of the set satisfies the predicate,
- no : the quantified expression is true if none of the elements of the set satisfy the predicate,
- at least <n> : the quantified expression is true if the predicate is satisfied for at least <n> elements of the set,
- at most <n> : the quantified expression is true if no more than <n> elements of the set satisfy the predicate, and
- exactly <n> : the quantified expression is true if the predicate is satisfied for exactly <n> elements of the set.

Given that support for quantified expressions of these kinds is already present in Daplex, the only new construct that is required is a new keyword to introduce a constraint declaration:

```
<constraint_dec>   ::= constrain <quant_part> <to_have> <predicate> ;
                    |  constrain <quant_part> <exists> ;

<quant_part>       ::= <quantifier> <named_set_expr> <quant_rest>
```

```
<quant_rest>       ::= so that <quant_part>
                    |  <quant_part>
                    |  []

<exists>           ::= exists | to exist | []
```

Here, <named_set_expr>, <predicate> and <to_have> are all defined by the existing Daplex language (in Appendix 2). Two constraint forms are defined, the first with an explicit predicate, e.g.

```
constrain each p in protein
to have molecular_weight(p) > 0;
```

and the second with an implied predicate of `true`, which is intended for use with the numerical quantifiers for placing restrictions on the cardinality of sets. E.g.

```
constrain each p in protein
so that some c in chains(p) exists;
```

which requires every protein in the database to have at least one constituent chain.

Since quantified expressions in Daplex are defined in terms of sets and predicates, all of the constructs provided by the language for describing these features are available for use in describing constraints. For example, here is a constraint from the protein structure database which uses the aggregate function `count` to ensure that each protein chain consists of the number of residues (i.e. smaller chemical groups which link together to form protein chains) that is recorded in its `num_residues` attribute:

```
constrain each c in chain
to have num_residues(c) = count(has_component_inv(c) as residue);
```

A fuller description of the semantics of the protein structure domain and the constraints which describe them can be found in (Embury 1995).

Quantifiers may be nested to an arbitrary degree, and may quantify over sets of arbitrary complexity, as is illustrated by the following constraint on the allowed chemical interactions (called "h-bonds") within helical segments of protein chain:

```
constrain each a in structure such that a is a alpha
    each h in hbond such that
        residue_structure(acceptor(h) as residue) = a and
        residue_structure(donor(h) as residue) = a and
        acceptor_atom(h) = "o" and
        donor_atom(h) = "n"
to have pos(donor(h)) = pos(acceptor(h)) + 3;
```

We can even use quantifiers in the set descriptions of other quantifiers. For example:

```
constrain each s in structure
        such that no s2 in structure has follows(s2) = s
to have first_structure(structure_chain(s)) = s;
```

This constraint is quantified over the `structure` class, the instances of which represent segments of protein chain which adopt recognisable 3-dimensional motifs such as helices and sheets. The constraint requires every structural segment which is not preceded by another segment to be the first structural segment of its chain.

Although our compiler transforms Daplex programs into Prolog procedures, Daplex itself is a functional language, with a semantics based on ZF-expressions (Paton 1990). ZF-expressions are a form of syntactic sugar for the λ-calculus (Landin 1966) which give a concise way to represent sets. For example, the following ZF-expression describes the set of large proteins, for which we gave the Daplex description earlier:

```
[ p | p <- protein; molecular_weight(p) > 2000]
```

The ZF-expression consists of a pattern (before the bar) and a list of generators, which produce potential values for the pattern variables, and restrictions on those values. Based on this idea, Daplex programs are converted into an intermediate code (ICode) format which also contains generators and restrictions:

```
[ generate(protein, var(evar1)),
  restrict(molecular_weight, [protein], [var(evar1)], var(evar2)),
  expression(>, var(evar2), 2000)
]
```

The semantics of the constraint language extension to Daplex, which are also based upon ZF-expressions, has been formalised for the CoLan language (Bassiliades, 1994), a predecessor and syntactic variant on the language described here. Internally, constraints take the same form as quantified expressions in the query language: the `restrict_subquery/3` construct. For example, the following fragment of ICode represents the constraint on the molecular weights of proteins given earlier:

```
restrict_subquery(all(var(evar1)),
    [generate(protein, var(evar1))],
    [restrict(molecular_weight, [protein], [var(evar1)], var(evar2)),
     expression(> var(evar2) 0)]
)
```

The first argument gives the quantification type and the quantified variable. The second argument describes the set of database values which are to be quantified over, while the third argument describes the predicate which must be applied to the set members. Constraints with multiple quantifiers are represented by nesting further `restrict_subquery/3` terms within the predicate description.

The reuse of so much of the existing Daplex syntax in the constraint language has obvious advantages for users of the database who are already familiar with Daplex, since it means that they do not have to learn a new language in order to be able to specify or understand constraints. Another advantage is that the clarity and readability of Daplex as a query language carries through to Daplex as a constraint language. This is, of course, helpful for the database designer who formulates the constraints, but we have found readability to be particularly valuable in discussions with domain experts, who must verify that the constraints are an accurate model of the application area's semantics but who are unfamiliar with the conventions of first order logic notation.

But perhaps the most significant advantage of the close integration of our constraint language into Daplex is that we have been able to reuse much of the existing compiler code in the implementation of the constraint language extension. The fact that we use the same internal representation for constraints as for other predicates has meant that we are able to use the current optimiser and code generator to process constraints, without requiring any extra programming. The parser, which converts the Daplex text into ICode, has also been reused almost in its original form, with the exception of five new parser rules to implement the extra syntax shown above. We have even been able to reuse the code generator when simplifying constraints, since this process has been implemented as a transformation from legal ICode to legal ICode. After simplification, the final translation to Prolog is handled by the current, unextended code generator. In a system which does not have a suitably expressive query language, all this work must be done from scratch for the constraint language.

4 THE P/FDM CONSTRAINT CHECKING ARCHITECTURE

The approach taken to constraint maintenance in P/FDM is essentially very simple. At compile-time, the declarative description of the constraint is analysed to see whether it is suitable for either of the two incremental checking methods available in the current version of the compiler: checking by simplified code, or checking by instance counting code. If it is not suitable for either of these methods, then we must use code which checks the entire database for validity. Code to perform this check is generated for all constraints, whatever their runtime checking behaviour will be, since it is used to check that the database into which a new constraint is to be asserted does not contain a violation. It is also used when a disabled constraint is re-enabled, which ensures that at any time all active constraints are satisfied by the database — the fundamental condition required for incremental constraint checking.

This code fragment (which we call the "initial code fragment") is generated by submitting the internal form of the constraint to the existing Daplex query language optimiser and code generator. As we saw in Section 3, constraints expressed in the extended Daplex language have exactly the same semantics, and therefore, internal representation, as quantified predicates in the original query language, and this allows us to reuse much of the original compiler code in the constraint language extension. In fact, for generating the initial code fragments, the original compiler can be re-used in its entirety. For example, the initial code fragment as generated by our current compiler from the constraint:

```
constrain each c in chain to have molwt(c) > 0;
```

is:

```
\+ (
     getentity(chain, A),
     getfnval(molwt, [A], B),
     B =< 0
).
```

The procedures `getentity/2` and `getfnval/3` called here are the database retrieval primitives. The `getentity/2` primitive takes the name of a database class as its first argument, and returns the identifiers of the instances of that class as its second argument, one by one on backtracking. The `getfnval/3` primitive takes the name of a database function as its first argument, and a list of the argument values to which it is to be applied as its second. The third argument returns the result of the function application. As with the entity retrieval primitive, multiple results from set-valued functions are returned individually, on backtracking. The code fragment generated from the above constraint, then, succeeds when no instances of the `chain` class can be found with an illegal, negative molecular weight, and fails when an invalid instance is found.

This example fragment was generated from a very simple constraint, but thanks to our reuse of the existing Daplex language compiler, we can handle more complex constraints just as easily. Here, for example, is the initial code fragment generated from our "hydrogen bonds in helices" constraint:

```
constrain each a in structure such that a is a alpha
    each h in hbond such that
        residue_structure(acceptor(h) as residue) = a and
        residue_structure(donor(h) as residue) = a and
        acceptor_atom(h) = "o" and
        donor_atom(h) = "n"
to have pos(donor(h)) = pos(acceptor(h)) + 3;
```

```
(
        getentity(structure, A),
        relative(alpha, A, B)
        atleast(1, (
             getentity(hbond, C),
             getfnval(acceptor, [C], D),
             relative(residue, D, E),
             getfnval(residue_structure, [E], F),
             F = A
             getfnval(donor, [C], G),
             relative(residue, G, H),
             getfnval(residue_structure, [H], I),
             I = A
             getfnval(acceptor_atom, [C], J),
```

```
                J = o
                getfnval(donor_atom, [C], K),
                K = n
                getfnval(pos, [G], L),
                getfnval(pos, [D], M),
                N is M + 3,
                L = N
            ))
        ).
```

Here, atleast/2 is a special predicate which takes an integer N and a fragment of Prolog code as its arguments, and which succeeds if the code fragment succeeds at least N times. The predicate will stop searching for further solutions once this limit has been reached. We can read this initial code fragment as a check that no alpha helix can be found for which some hydrogen bond within that helix links two residues that are either closer to or further from each other than the regulation four residues.

The generation of the initial code fragment for constraints which will be checked by simplified code is exactly the same as for those which cannot be simplified. However, constraints which are to be checked using the instance counter method must be treated slightly differently. For this kind of constraint, we need to know not only that the constraint is satisfied by the database, but also for how many instances it is satisfied, so that we can initialise the constraint counter correctly. In this case, we merely exchange the quantified expression for an application of the count aggregate function (to count the number of times the predicate is satisfied) followed by an expression to check that the result does not exceed the required limit. For example, a constraint whose internal representation is:

```
restrict_subquery(atmost(Var, Num), SetCode, PredCode)
```

would be translated into:

```
[ expression(count, ResultVar, SolutionVar, [logical_and(SetCode, PredCode)]),
  expression(=<, ResultVar, Num)
]
```

The initial code fragment is then generated using the standard code generator as:

```
findall(SolutionVar, (SetCode, PredCode), SolutionList),
count(SolutionList, ResultVar),
ResultVar =< Num
```

The same basic framework is used to generate code for constraints quantified by the at_least and exactly quantifiers.

Once the initial code has been generated, and used to prove that the new constraint is satisfied by the current database, it is stored in the metadata, along with an explicit two-way link to the metadata of the new constraint.

The next stage of the compilation process involves the identification of the set of events which may violate the constraint, after the occurrence of which it must be checked. It is obviously important that a minimal, or near-minimal, set of events is generated, since this will reduce the amount of times the constraint checking code has to be executed. Our event detection algorithm, which is described in detail in Section 5, generates the minimal event set in the majority of cases, and a near minimal set for a few particular constraint types.

The set of events generated for each constraint is used to build a two-way index in the metadata, linking constraints and the events which may violate them. This index is used primarily for discovering the set of constraints which must be checked when a particular update occurs, but it has also been found to be useful when performing semantic query optimisation, in identifying the constraints which may constructively be added to a data manipulation program (Embury 1994).

Finally, for those constraints which are suitable for incremental checking, the simplified code fragments are generated (as we'll describe in Section 6). One or more code fragments are generated for each of the potentially violating update events, and a further metadata link is created, to preserve the relationship between the constraint and all the simplified code fragments generated from it, for use by the run-time constraint checker.

At the end of the compilation process, the metadata will contain at least three representations of the new constraint (constraints which can be simplified will have an extra representation), all linked by two-way indexes. These various representations are:

- the original Daplex specification of the constraint, stored as a Prolog atom,
- the internal representation of the constraint, as an ICode restrict_subquery/3 term,
- the initial code fragment, which forms a procedural representation of the entire constraint, and
- the simplified constraint fragments, which form an additional procedural representation of the constraint.

Both the Daplex and the ICode versions of the declarative constraint are retained as the former is suitable for displaying to the user, when giving feedback on violated constraints, for example, while the latter is more suitable for manipulation by other programs, such as a semantic query optimiser.

The run-time behaviour of the P/FDM constraint checker is specified as a guard to each database update primitive. This guard uses the metadata indexes generated at compile-time, and the details of the current update to decide which constraints (including inherited constraints) may be violated by the update, and consequently which code fragments must be executed. These fragments will either be initial code fragments, for those constraints which cannot be checked incrementally, or the appropriate simplified code fragment for the given event. If the update passes all the checks then it is allowed to proceed.

One advantage of this approach to constraint checking is that the guard (i.e. the constraint checker) decides which code fragments must be executed independently of any other system binding mechanism (such as method binding), at run-time. This means that constraints may be added to or deleted from the schema without requiring any recompilation of methods or other application code. It also allows us to specialise the inheritance strategy for constraints according to the semantics we require. In P/FDM, for example, we have implemented a form of inclusion inheritance for constraints, where all constraints from all superclasses are inherited, regardless of whether the function they constrain has been specialised or not, while our method inheritance mechanism has the standard overriding semantics.

This run-time retrieval of code fragments imposes some extra overhead on constraint checking, although our use of metadata indexes keeps this relatively low and independent of the number active constraints (see Appendix 1). As far as efficiency of the constraint checking process as a whole is concerned, however, the main advantage of our architecture is that it allows the complete modelling of the many-to-many relationship between events and constraint code fragments, which as we saw in Section 2 is crucial to achieving a high degree of simplification.

5 IDENTIFICATION OF POTENTIALLY VIOLATING EVENTS FOR ARBITRARY CONSTRAINTS

As we have seen, the efficient compilation of a declaratively specified constraint depends upon the identification of a near-minimal set of events which may potentially violate it, as well as a thorough simplification technique. Given a range-restricted constraint language, it is easy to generate a first approximation to the set of potentially violating events, simply by enumerating each class and function mentioned in the constraint and creating the set of all possible update operations that may be applied to each one. In P/FDM, there are three update primitives for operating on classes:

`newentity(+Class, +Key, -Id)` : creates a new instance of `Class`, with the given `Key` and returns its identifier as `Id`,

`include(+Class, +OldInst, -NewInst)` : extends the instance hierarchy of which `OldInst` is a member so that it includes `Class`. (`OldInst` must be a member of a class in the same hierarchy as `Class`, although it need not be a member of a superclass of `Class`). The identifier of the new `Class` instance is returned as `NewInst`,

`deletentity(+Class, +InstId)` : deletes the given instance of the given class.

Three primitives for updating functions are also provided:

`addfnval(+FName, +Arguments, +Result)` : creates a mapping from `Arguments` to `Result` for the function called `FName`,

`deletefnval(+FName, +Arguments, +Result)` : deletes the mapping from `Arguments` to `Result` from the function called `FName`, and

`updatefnval(+FName, +Arguments, +OldResult, +NewResult)` :
replaces `OldResult` with `NewResult` in the mapping from `Arguments` for the function called `FName`.

To illustrate the generation of this first approximation to the set of potentially violating events for this constraint, consider the following example constraint from the protein database:

```
constrain each r in residue
to have pos(r) >= start(residue_structure(r)) and
    pos(r) =< end(residue_structure(r));
```

The maximum set of potentially violating events for this constraint are:

```
newentity(residue)
include(residue)
deletentity(residue)
newentity(structure)
include(structure)
deletentity(structure)
addfnval(start(structure))
deletefnval(start(structure))
updatefnval(start(structure))
addfnval(end(structure))
deletefnval(end(structure))
updatefnval(end(structure))
addfnval(pos(residue))
deletefnval(pos(residue))
updatefnval(pos(residue))
addfnval(residue_structure(residue))
deletefnval(residue_structure(residue))
updatefnval(residue_structure(residue))
addfnval(residue_structure_inv(residue))
deletefnval(residue_structure_inv(residue))
updatefnval(residue_structure_inv(structure))
```

Notice that we have included the three possible events on the `structure` class in the set, even though it is not mentioned directly in the constraint. It is present here as an intermediate class in the composed function applications of `start(residue_structure(r))` and `end(residue_structure(r))`, and must therefore be considered when searching for potentially violating events. Intermediate classes are easily generated as they are identified uniquely by the combination of the argument type of the first and the result type of the second of the composed functions. Notice also that we have included updates to the system generated inverse of the `residue_structure` function, `residue_structure_inv`, in the set of events. Inverse functions are generated for all relationship functions (i.e. those which map an entity class onto an entity class) and are maintained automatically by the function update primitives, which propagate all changes to functions to their inverses, and *vice versa*. Hence the inclusion of updates to inverse functions within the set of potentially violating events.

However, this set of potentially violating events will contain some events which cannot under any circumstances violate the constraint being analysed. For example, the `deletentity(residue)` event cannot possibly violate the above example constraint, since once an instance ceases to be a member of the `residue` class, it ceases to be bound by the constraint. In a case like this, we say that the constraint is "trivially satisfied" by the event, since no processing effort is required to decide whether the update will preserve the database's integrity. In fact, the proportion of trivially satisfying events to potentially violating events can be quite high. Typically, just under half of the events in the maximal event set will be trivially satisfying. Our aim, then, in generating the set of potentially violating events for a given constraint is to identify only those events which may actually violate the constraint, and to include as few events for which the constraint is trivially satisfied as possible.

Unfortunately, this process is complicated by the fact that each different quantifier and predicate type has its own semantics and is violated by its own combination of events. The `deletentity(residue)` event, for example, is only trivially satisfying for the example constraint because of the presence of the universal quantifier. If the same constraint is existentially quantified, then the deletion of a `residue` instance may cause a violation, since it may delete the only `residue` instance which satisfies the constraint's predicate.

Another factor which influences the set of potentially violating events for a given quantifier is whether it is interpreted under a strong- or a weak-translation semantics (Urban 1989). A weak-translation of a predicate is trivially satisfied if any of the attributes involved in it are undefined. The following constraint, for example, requires only that all proteins *whose molecular weight is known* weigh more than 0:

```
constrain each p in protein to have molecular_weight(p) > 0;
```

Under the alternative translation semantics, i.e. a strong-translation, the constraint would require that, for all `protein` instances, the `molecular_weight` function is defined and that its value falls within the required range. The difference between these two semantics is best illustrated by their equivalent first order logic expressions (hence the terms weak- and strong-*translation*). A weak-translation to FOL gives:

$$(\forall p, w)\ (protein(p)\ \wedge\ molecular\_weight(p, w))\ \Rightarrow\ w > 0$$

while the strong-translation gives:

$$(\forall p)\ protein(p)\ \Rightarrow\ ((\exists w)\ molecular\_weight(p, w)\ \wedge\ w > 0)$$

In P/FDM, we adopt a weak-translation for universally quantified constraints since it is the most flexible and the most practical of the two possibilities; flexible because it allows us to constrain both total and partial functions without changing their semantics (i.e. the partial functions do not become total functions when constrained), and practical because it allows us to constrain classes and functions which are only partially defined. For example, under a strong-translation semantics, a constraint such as that given above could only be defined when the constrained attributes of all the instances were defined, while in real world situations it often occurs that we do not have all the data that we require at data definition time.

For existentially quantified constraints, on the other hand, we take a strong-translation semantics, which more accurately reflects the intended semantics of this kind of constraint. For example, consider the constraint that some large protein must exist:

```
constrain some p in protein to have molecular_weight(p) > 2000;
```

Under a strong translation, this constraint is only satisfied if some protein is known to have a high molecular weight. Under a weak translation, it is also true if some protein exists for which the weight is not defined, which is counter-intuitive and which complicates the process of reasoning with existentially quantified constraints.

We can characterise the two types of translation semantics according to the effect that the deletion of data involved in a predicate (whether an instance or a function value) will have on its truth value. Under a strong translation, a deletion event will always falsify a true predicate, and will have no effect upon a false predicate. Conversely, under a weak translation, a deletion event always satisfies a false predicate, and has no effect on the truth of a satisfied predicate. According to this characterisation, we can ignore any deletion events on a weakly-translated predicate, since they cannot possibly cause it to become false for the values involved in the deletion. For the constraint on residues given earlier, this means that we can remove all the deletion events from the maximal set of twenty-one events, giving a reduced set of just fourteen events.

In fact, we can use the fact that this constraint has a weak translation semantics to remove the two `newentity` and the two `include` events from the potentially violating set, on the grounds that neither of these updates can create an instance for which all the functions mentioned in the constraint are populated. A `newentity` update, for example, creates an instance whose key functions are all guaranteed to be populated, and whose non-key attributes are all guaranteed to be undefined. In our example constraint, of the two functions on the `residue` class that are mentioned in the constraint predicate, only the `pos` key function will have a value after a `newentity` event, while the `residue_structure` function will be undefined. And of the three functions on the `structure` class (`start`, `end` and `residue_structure_inv`) none are key functions and therefore all will be undefined after the creation of a new `structure` instance. Clearly, then, under the weak translation semantics of the universal quantifier, neither `newentity` event can violate the constraint.

The situation with the `include` events is analogous, in that this type of update creates an instance for which the key functions are guaranteed to be defined, the other attribute functions which are inherited by the new instance may or may not be defined and the attribute functions defined directly on the instance are guaranteed not to be defined. Therefore, an `include` update event trivially satisfies a constraint if and only if that constraint does not include any non-inherited functions on the included class, since these functions will all be undefined. Accordingly, we know that the `include(residue)` event trivially satisfies the example constraint since, although the `pos` attribute is inherited, the `residue_structure` function is not. We can also disregard the `include(structure)` event, since `structure` is a root class, which inherits no functions (not even its key functions) and for which an inclusion update is meaningless.

The set of potentially violating events for this constraint is now reduced to an `addfnval` and an `updatefnval` event for each of the five functions involved in the constraint predicate — a total of ten events. In fact, we can remove just one more event — the `addfnval` event on the `pos` function — since this is a key function, for which `addfnval` is an invalid update. The same is also true of `deletefnval` events on key functions, which are not allowed because they would cause instances to exist with partial keys. The `updatefnval` primitive is allowed to be applied to key functions, however, provided that the update does not violate the uniqueness constraint on key values.

The set of potentially violating events for this class now consists of the minimal set of nine events:

```
updatefnval(pos(residue))        addfnval(residue_structure(residue))
addfnval(start(structure))       updatefnval(residue_structure(residue))
updatefnval(start(structure))    addfnval(residue_structure_inv(structure))
addfnval(end(structure))         updatefnval(residue_structure_inv(structure))
updatefnval(end(structure))
```

These rules apply only to predicates which are governed by a quantifier with a weak translation semantics. For the strongly-translated numerical quantifiers, we are concerned only with updates which can cause the number of times the constraint predicate is satisfied to move outside the range prescribed by the constraint. Consider, for example, the following constraint:

```
constrain at least 5 p in protein
```

```
to have molecular_weight(p) > 2000;
```

which requires the protein database to contain no fewer than five large proteins. The maximal set of potentially violating events for this constraint is given below, divided into those events which may reduce the number of times the constraint predicate is satisfied, and those which may increase the number of satisfying protein instances:

Reducing	Increasing
`deletentity(protein)`	`newentity(protein)`
`deletefnval(molecular_weight(protein))`	`include(protein)`
`updatefnval(molecular_weight(protein))`	`addfnval(molecular_weight(protein))`
	`updatefnval(molecular_weight(protein))`

We have included the `updatefnval(molecular_weight(protein))` event twice here since it is capable both of reducing and increasing the number of times the predicate is satisfied. From this set, we can immediately discount the `include(protein)` event, since `protein` is a root class, and the `newentity(protein)` event, since the `molecular_weight` function is not a key attribute of `protein` and therefore this update cannot create a new instance which satisfies the (strongly-translated) predicate.

The `at least` numerical quantifier imposes a lower limit on the number of times the constraint predicate may be satisfied by a database state. Clearly, then, given an initial database state in which the constraint is satisfied, any event which reduces this count may potentially violate it, but any event which increases the count cannot, under any circumstances, cause it to fall below the specified lower limit and cannot therefore cause the constraint to be violated. This kind of constraint is said to have a "reducing" semantics.

Similarly, a constraint which is quantified by the `at most` quantifier cannot be violated by any event which reduces the number of times the constraint predicate is satisfied, but may be violated by any event which increases the number of values for which it is satisfied. This semantics is called an "increasing" semantics. Constraints quantified by the `exactly` quantifier can be violated by any event which causes a change in the number of times the constraint is violated, i.e. they have both an increasing and a reducing semantics.

For the majority of constraints, the trivially satisfying events can be ignored and do not require any further code generation. However, for those constraints which are suitable for checking by the instance counting method, we have to ensure that the count of times the constraint predicate is satisfied is kept up to date at all times. This means that code fragments must be generated for all events which may cause a change in the counters value, even if that change cannot violate the constraint itself. Fortunately, the same basic algorithm, which we shall now describe, can be used to generate both the set of potentially violating events and, when required, the set of count altering events.

5.1 The Event Generation Rules

The above rules for identifying the minimal set of potentially violating events for a given constraint have been formalised in terms of the intermediate code format of our constraint language. The complete algorithm deals with twenty-one different intermediate code constructs, so for brevity we give only a subset of the rules here. In what follows,

c is the constraint under consideration, which is assumed to be satisfied by the initial database state, $E^{Sem}_{fal}(p)$ is the set of events which will potentially falsify the true predicate p, under the translation semantics Sem, and $E^{Sem}_{sat}(p)$ is the set of events which will potentially satisfy the false predicate p, under the translation semantics Sem. All of the four different semantics (weak, strong, reducing and increasing) must be considered. When the behaviour of some predicate is the same under all semantics then Sem is given as $\_$, e.g. $E^{-}_{fal}(p)$.

The overall aim of this analysis is to identify the set $E^{-}_{fal}(c)$ — the set of potentially violating events for the entire constraint. In general, this will be the minimal event set; the cases in which a near-minimal set is generated are outlined below. For the numerical quantifiers, we may also need to identify the set $E^{-}_{alt}(c)$, which is the set of events which may alter the number of times the constraint is satisfied. The rules for generating this set are given with the rules for the appropriate quantifiers.

A universally quantified predicate

$$E^{-}_{fal}(restrict\_subquery(all(\_), set, predicate) \equiv E^{strong}_{sat}(set) \cup E^{weak}_{fal}(predicate)$$

$$E^{-}_{sat}(restrict\_subquery(all(\_), set, predicate) \equiv E^{weak}_{fal}(set) \cup E^{strong}_{sat}(predicate)$$

A universally quantified constraint may potentially be falsified by any event which creates a new set member for which the predicate may be false ($E^{strong}_{sat}(set)$), or any event which falsifies the predicate for some existing set member ($E^{weak}_{fal}(predicate)$). It cannot be falsified by any event which causes some instance to no longer be a member of the quantified set, or any event which causes the predicate to become satisfied for some value.

Conversely, a universally quantified constraint which is not satisfied in a particular database state may be resatisfied by any event which causes the offending set member (i.e. the value for which the predicate is not satisfied) to be removed from the constraint set, or by any event which resatisfies the predicate for the offending set member.

An existentially quantified predicate

$$E^{-}_{fal}(restrict\_subquery(some(\_), set, predicate) \equiv E^{red}_{fal}(set) \cup E^{red}_{fal}(predicate)$$

$$E^{-}_{sat}(restrict\_subquery(some(\_), set, predicate) \equiv E^{inc}_{sat}(set) \cup E^{inc}_{sat}(predicate)$$

An existentially quantified predicate which is satisfied by a database state can be falsified by any event which reduces the number of instances which are members of the set quantified over by the predicate, or by any event which can reduce the number of set members for which the predicate is satisfied. A false existentially quantified predicate can be resatisfied by any event that adds a new set member which satisfies the predicate, or by any event which causes an existing set member to resatisfy the predicate.

Since existentially quantified constraints are candidates for the "instance count" method of incremental checking, we must also define the set E_{alt} for this type of predicate. This set can be easily determined by treating the constraint as if it were an "exactly" quantified constraint:

$$E^{-}_{alt}(restrict\_subquery(some(v), set, predicate) \equiv E^{-}_{fal}(restrict\_subquery(exactly(v, 1), set, predicate))$$

A negated existentially quantified predicate

$$E^{-}_{fal}(restrict\_subquery(no(\_), set, predicate) \equiv E^{inc}_{sat}(set) \cup E^{inc}_{sat}(predicate)$$

$$E^{-}_{sat}(restrict\_subquery(no(\_), set, predicate) \equiv E^{red}_{fal}(set) \cup E^{red}_{sat}(predicate)$$

A predicate which is quantified by a "not exists" quantifier can be falsified by any event which causes a new instance to become a member of the set quantified over (and which satisfies the predicate), or by any event which causes an existing set member to satisfy the predicate. We do not need to worry about events which reduce the number of times this kind of constraint is satisfied, since if we know that the constraint is initially true then we know that it is satisfied zero times.

To resatisfy a false predicate quantified in this way, an update that will reduce the number of times it is satisfied must occur.

A numerically quantified predicate specifying a lower limit

$$E^{-}_{fal}(restrict\_subquery(atleast(\_,\_), set, predicate) \equiv E^{red}_{fal}(set) \cup E^{red}_{fal}(predicate)$$

$$E^{-}_{sat}(restrict\_subquery(atleast(\_,\_), set, predicate) \equiv E^{inc}_{sat}(set) \cup E^{inc}_{sat}(predicate)$$

This type of predicate has the same set of potentially violating events as an existentially quantified predicate. The "*exists*" quantifier can be seen as a special case of the "*at least N*" quantifier, where $N = 1$.

Similarly, we can give the same definition for the set E_{alt} for this type of predicate as for existentially quantified predicates:

$$E^{-}_{alt}(restrict\_subquery(atleast(v, n), set, predicate) \\ \equiv E^{-}_{fal}(restrict\_subquery(exactly(v, n), set, predicate))$$

A numerically quantified predicate specifying an upper limit

$$E^{-}_{fal}(restrict\_subquery(atmost(\_,\_), set, predicate) \;\equiv\; E^{inc}_{sat}(set) \;\cup\; E^{inc}_{sat}(predicate)$$

$$E^{-}_{sat}(restrict\_subquery(atmost(\_,\_), set, predicate) \;\equiv\; E^{red}_{fal}(set) \;\cup\; E^{red}_{fal}(predicate)$$

Predicates quantified by the "*at most N*" quantifier have the same set of potentially violating events as predicates quantified by "*not exists*", which can be seen as a special case of this quantifier where $N = 0$. However, they can also be checked by the "instance count" method, and we must therefore generate the set of count altering events. Again, we can derive this set by considering the constraint as if it were quantified by the "exactly" quantifier:

$$E^{-}_{alt}(restrict\_subquery(atleast(v,n), set, predicate) \equiv E^{-}_{fal}(restrict\_subquery(exactly(v,n), set, predicate))$$

An exactly quantified predicate

$$E^{-}_{fal}(restrict\_subquery(exactly(\_,\_), set, predicate) \equiv E^{red}_{fal}(set) \cup E^{inc}_{sat}(set) \cup E^{red}_{fal}(predicate) \cup E^{inc}_{sat}(predicate)$$

$$E^{-}_{sat}(restrict\_subquery(exactly(\_,\_), set, predicate) \equiv E^{inc}_{sat}(set) \cup E^{red}_{fal}(set) \cup E^{inc}_{sat}(predicate) \cup E^{red}_{fal}(predicate)$$

A predicate which is quantified by the "*exactly N*" quantifier can be falsified by any event which changes the number of set members for which the predicate is satisfied, either by removing a set member, creating a new set member which satisfies the constraints, causing some existing set member to satisfy the predicate or causing the predicate to become false for some existing set member. Any of these events can also potentially resatisfy an exactly quantified constraint.

A logical conjunction of predicates

$$E^{Sem}_{fal}(logical\_and(lhs, rhs)) \;\equiv\; E^{Sem}_{fal}(lhs) \;\cup\; E^{Sem}_{fal}(rhs)$$

$$E^{Sem}_{sat}(logical\_and(lhs, rhs)) \;\equiv\; E^{Sem}_{sat}(lhs) \;\cup\; E^{Sem}_{sat}(rhs)$$

A logical disjunction of predicates

$$E^{Sem}_{fal}(logical\_or(lhs, rhs)) \;\equiv\; E^{Sem}_{fal}(lhs) \;\cup\; E^{Sem}_{fal}(rhs)$$

$$E^{Sem}_{sat}(logical\_or(lhs, rhs)) \;\equiv\; E^{Sem}_{sat}(lhs) \;\cup\; E^{Sem}_{sat}(rhs)$$

A logically negated predicate

$$E^{Sem}_{fal}(logical\_not(predicate)) \;\equiv\; E^{Sem}_{sat}(predicate)$$

$$E_{sat}^{Sem}(logical\_not(predicate)) \;\equiv\; E_{fal}^{Sem}(predicate)$$

A comparison expression

$$E_{fal}^{-}(expression(compop, value1, value2)) \;\equiv\; \{\}$$

$$E_{sat}^{-}(expression(compop, value1, value2)) \;\equiv\; \{\}$$

Comparison operators currently supported are: `=`, `<>`, `>`, `>=`, `<` and `=<`. The truth value of a comparison expression is not directly altered by an update, although it may be indirectly altered by updates which change the values being compared. However, these indirectly violating events are generated from the description of the values to be compared (given later) and not from the comparison expression itself. Hence the empty sets generated by these rules.

The following rules describe the generation of events from ICode constructs which are not predicates in themselves, but whose value contributes to the truth value of some predicate.

A set operator application

$$E_{fal}^{-}(generate\_subquery(setop, set1, set2)) \\ \equiv E_{fal}^{strong}(set1) \;\cup\; E_{fal}^{strong}(set2) \;\cup\; E_{sat}^{weak}(set1) \;\cup\; E_{sat}^{weak}(Set2)$$

$$E_{sat}^{-}(generate\_subquery(setop, set1, set2) \\ \equiv E_{sat}^{weak}(set1) \;\cup\; E_{sat}^{weak}(set2) \;\cup\; E_{fal}^{strong}(set1) \;\cup\; E_{fal}^{strong}(set2)$$

Three set operators are provided by Daplex: union, intersection and difference. In all cases, an expression involving a set operator can alter the truth value of the primitive in which it is involved if the set it describes is altered to contain extra or fewer elements. This in turn can be caused by any event which alters the contents of the two sets to which the operator is applied.

An aggregate operator application

$$E_{fal}^{-}(expression(aggop, set)) \;\equiv\; E_{fal}^{strong}(set) \;\cup\; E_{sat}^{weak}(set)$$

$$E_{sat}^{-}(expression(aggop, set) \;\equiv\; E_{sat}^{weak}(set) \;\cup\; E_{fal}^{strong}(set)$$

Five aggregate operators are provided in the current version of the Daplex language: count, total, minimum, maximum and average. Any change to the cardinality or membership of the set to which the aggregate is applied can cause a change to the final result, and thereby a change to the truth value of the predicate in which it is involved.

However, it is in this case that a near-minimal and not the minimal set of potentially violating events is generated, since we do not as yet take full advantage of our knowledge of the semantics of these operators. For example, if we have an application of the `count` operator to some set, we know that any operation which increases the cardinality of the set will increase the result of the aggregation, and any update which decreases the set's cardinality will decrease the result value. We could therefore disregard all events which would increase the cardinality of *set* in predicates of the form:

```
value < count(set)
```

and all events which decrease the cardinality for events of the form:

```
value > count(set)
```

Similarly, we know that removing values from a set can never cause its maximum element to have a higher value than the maximum element prior to the update, and *vice versa* for minimum elements. We have not yet incorporated this kind of knowledge into our event detection algorithm because we have not yet found the aggregate operators to be particularly useful in describing the semantics of our protein structure domain. If, however, this kind of construct did begin to appear to be more common (perhaps in another domain area) then this extra semantic knowledge could be included into the algorithm.

The remainder of the rules concern the class and function value enumeration constructs (i.e. the generators and restrictions of the ICode set expressions), from which the actual events are generated. The events are described by the following sets:

- *fs(c)* which is the set of pairs of all functions and the variables and values to which they are applied in the constraint *c*,
- *cs(c)* which is the set of pairs of all instance variables and their classes present in the constraint *c*,
- *DE(class)* $\equiv$ {*event(deletentity, class)*} which generates a deletion event descriptor for the given class,
- *NE(class, var)* $\equiv$ {*event(newentity, class)*}
  iff $((\exists f_1, as_1)\ (f_1,\ as_1)\ \epsilon\ fs(c) \wedge var\ \epsilon\ as_1) \wedge$
  $(\neg\ (\exists\ f_2,\ as_2)\ (f_2,\ as_2)\ \epsilon\ fs(c) \wedge var\ \epsilon\ as_2 \wedge \neg\ key\_function(f_2,\ class))$
  $\equiv$ {} otherwise.
  This rule states that a *newentity* event is only generated for a reference to the given *class* as the variable *var* if some function f_1 is applied to *var* within the constraint, but that no non-key function is applied to *var*,
- *IE(class, var)* $\equiv$ { *event(include, class)* }
  iff $\neg superclass(class,\ entity) \wedge$
  $(\neg\ (\exists\ f,\ as)\ (f,\ as)\ \epsilon\ fs(c) \wedge var\ \epsilon\ as \wedge defined\_on(f,\ class))$
  $\equiv$ {} otherwise.
  An *include* event is only generated by a reference to a class (by a variable *var*) if the class is not a root-class (that is, if it's superclass is not the top-level entity class *entity*) and if none of the functions applied to *var* within the constraint are defined directly on the class. Notice that, unlike *newentity* events, we do not require there to be at least one

function applied to *var* in the constraint in order to generate the *include* event. This is because we sometimes test for membership of a subclass without applying any further predicates to the subclass instance. The constraint on hydrogen bonds in helices given on page 7 illustrates this kind of test — an event which includes an existing `structure` instance into the `alpha` class may violate this constraint, even though no functions are applied to the `alpha` instance.

- *AE(fname, argtype)* $\equiv \{\}$
  iff *key_function(fname, argtype)*
  $\equiv \{$*event(addfnval, fname, argtype), event(addfnval, inv, restype)*$\}$
  iff *has_inverse(fname)* $\wedge$ *inv_name(fname, inv)* $\wedge$
  result_type(fname, restype)
  $\equiv \{$*event(addfnval, fname, argtype)*$\}$ otherwise.

  Addfnval events are only generated for non-key functions. If the function has an inverse then an update event must be generated for that function also.
- *FE(fname, argtype)* $\equiv \{\}$
  iff *key_function(fname, argtype)*
  $\equiv \{$*event(deletefnval, fname, argtype),*
  event(deletefnval, inv, restype)$\}$
  iff *has_inverse(fname)* $\wedge$ *inv_name(fname, inv)* $\wedge$
  result_type(fname, restype)
  $\equiv \{$*event(deletefnval, fname, argtype)*$\}$ otherwise.

  Deletefnval events are only generated for non-key functions, and for inverse functions, where they exist.
- *UE(fname, argtype)* $\equiv \{$*event(updatefnval, fname, argtype),*
  event(updatefnval, inv, restype)$\}$
  iff *has_inverse(fname)* $\wedge$ *inv_name(fname, inv)* $\wedge$
  result_type(fname, restype)
  $\equiv \{$*event(updatefnval, fname, argtype)*$\}$ otherwise.

  Updatefnval events are generated for both key and non-key functions, and for inverse functions, where they exist.

These sets are used to describe the events generated from the remaining ICode constructs under the various translation semantics.

A class enumerating generator

$$E_{fal}^{weak}(generate(class, var)) \equiv NE(class, var) \cup IE(class, var)$$

$$E_{fal}^{strong}(generate(class, var)) \equiv NE(class, var) \cup IE(class, var) \cup DE(class)$$

$$E_{fal}^{red}(generate(class, var)) \equiv DE(class)$$

$$E_{fal}^{inc}(generate(class, var)) \equiv NE(class, var) \cup IE(class, var)$$

$$E_{sat}^{weak}(generate(class, var)) \equiv NE(class, var) \cup IE(class, var) \cup DE(class)$$

$$E_{sat}^{strong}(generate(class, var)) \equiv NE(class, var) \cup IE(class, var)$$

$$E^{red}_{sat}(generate(class, var)) \equiv NE(class, var) \cup IE(class, var)$$

$$E^{inc}_{fal}(generate(class, var)) \equiv DE(class)$$

A cast operation on a class instance

$$E^{Sem}_{fal}(relative(class, \_, var)) \equiv E^{Sem}_{fal}(generate(class, var))$$

$$E^{Sem}_{sat}(relative(class, \_, var)) \equiv E^{Sem}_{sat}(generate(class, var))$$

A function result generator

$$E^{weak}_{fal}(generate(fname, argtype, \_, \_, \_)) \equiv AE(fname, argtype) \cup UE(fname, argtype)$$

$$E^{strong}_{fal}(generate(fname, argtype, \_, \_, \_)) \equiv AE(fname, argtype) \cup FE(fname, argtype) \cup UE(fname, argtype)$$

$$E^{red}_{fal}(generate(fname, argtype, \_, \_, \_)) \equiv FE(fname, argtype) \cup UE(fname, argtype)$$

$$E^{inc}_{fal}(generate(fname, argtype, \_, \_, \_)) \equiv AE(fname, argtype) \cup UE(fname, argtype)$$

$$E^{weak}_{sat}(generate(fname, argtype, \_, \_, \_)) \equiv AE(fname, argtype) \cup FE(fname, argtype) \cup UE(fname, argtype)$$

$$E^{strong}_{sat}(generate(fname, argtype, \_, \_, \_)) \equiv AE(fname, argtype) \cup UE(fname, argtype)$$

$$E^{red}_{sat}(generate(fname, argtype, \_, \_, \_)) \equiv AE(fname, argtype) \cup UE(fname, argtype)$$

$$E^{inc}_{fal}(generate(fname, argtype, \_, \_, \_)) \equiv FE(fname, argtype) \cup UE(fname, argtype)$$

A function value restriction

$$E^{Sem}_{fal}(restrict(fname, argtype, \_, \_)) \equiv E^{Sem}_{fal}(generate(fname, argtype, \_, \_, \_))$$

$$E^{Sem}_{sat}(restrict(fname, argtype, \_, \_)) \equiv E^{Sem}_{sat}(generate(fname, argtype, \_, \_, \_))$$

These rule sets describe the complete algorithm for generating the set of potentially violating events for any Daplex predicate of arbitrary complexity. They have been transcribed directly into the Prolog rules which perform this function for the constraint language compiler. The advantage of this algorithm is that it operates by examining the ordinary ICode representation of the constraint, unlike other event detection methods which have been proposed and which require the constraint to be converted into Horn clauses (Urban 1989, Díaz 1992). Our algorithm can also be easily extended to include new language features, simply by adding new rules to generate E_{fal} and E_{sat} for the new constructs.

6 GENERATION OF SIMPLIFIED CONSTRAINT CHECKING CODE

Having identified the events that may potentially cause a violation of a particular constraint, it is necessary to generate the code fragments that will identify a definite violation for each such event. The aim is to reduce the amount of processing that must be done at run-time, by using our knowledge of the semantics of the update events and the constraint to partially evaluate it at compile-time.

Our current compiler is able to simplify only a limited subset of the constraints which are expressible in our extended Daplex language, although we hope to extend this subset in the future. The subset which is simplifiable is the set of constraints which have a single "forall", "not exists" or "exactly" quantifier, operating over an unrestricted database class, and applying a conjunction of comparisons between single values as its predicate. Or, to put this more formally, a constraint c is simplifiable if:

$$\begin{aligned}
&\mathrm{c} \equiv \mathrm{restrict_subquery(q,\ s,\ p)}\\
&\quad\wedge\ \mathrm{q}\ \epsilon\ \{\mathrm{all,\ no,\ exactly}\} \wedge \mathrm{s} = [\mathrm{generate(_,\ _)}] \wedge \mathrm{simplifiable_predicate(p)}
\end{aligned}$$

where

$$\begin{aligned}
&\mathit{simplifiable_predicate(logical_and(lhs,\ rhs))}\\
&\qquad\qquad \equiv \mathit{simplifiable_predicate(lhs)} \wedge \mathit{simplifiable_predicate(rhs)}
\end{aligned}$$

$$\begin{aligned}
&\mathit{simplifiable_predicate(expression(compop,\ _,\ _))}\\
&\qquad\qquad \equiv \mathit{compop}\ \epsilon\ \{=,\ <>,\ <,\ =<,\ >,\ >=\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{simplifiable_predicate(p)}\\
&\qquad\qquad \equiv \mathit{restrict(fname,\ argtype,\ args,\ _)}\ \epsilon\ p\\
&\qquad\qquad\quad \wedge\ \mathit{single_valued_function(fname,\ argtype)}\\
&\qquad\qquad\quad \wedge\ \mathit{simplifiable_predicate(p\ \text{-}\ \{restrict(fname,\ argtype,\ args,\ _)\})}
\end{aligned}$$

$$\begin{aligned}
&\mathit{simplifiable_predicate(p)}\\
&\qquad\qquad \equiv \mathit{generate(fname,\ argtype,\ args,\ _,\ _)}\ \epsilon\ p\\
&\qquad\qquad\quad \wedge\ \mathit{single_valued_function(fname,\ argtype)}\\
&\qquad\qquad\quad \wedge\ \mathit{simplifiable_predicate(p\ \text{-}\ \{generate(fname,\ argtype,\ args,\ _,\ _)\})}
\end{aligned}$$

The compiler is also able to provide more efficient checking for another class of constraints, namely those which are quantified by a single "exists", "at least" or "at most" quantifier, and which otherwise have the same form as the simplifiable constraints described above. For constraints of this sort, we generate simplified code fragments which will detect a change in the number of times the constraint is satisfied and check whether this change will violate the constraint. If the constraint is not violated, then the count of the number of times the predicate is satisfied is updated. Formally, the set of constraints which are checked by this method is the set of constraints c for which:

$$c \equiv restrict\_subquery(q,\ s,\ p)$$
$$\wedge\ q \in \{some,\ atleast,\ atmost\} \wedge s = [generate(\_,\ \_)] \wedge simplifiable\_predicate(p)$$

where the definition of *simplifiable_predicate* is that given above.

The same basic simplification algorithm is used to generate code fragments for both these types of constraints, the difference being in the response to the success or failure of the constraint checking code. In all cases, the compiler generates one or more Prolog clauses for each potentially violating event. Each clause has a unique functor, and one or two arguments which pass the context of the update to the constraint checking code. The body of the clause succeeds if the constraint will be violated by the update, and fails if it may proceed. We will show some complete constraint checking clauses that have been generated by our compiler later in this section, after we have described the simplification process itself.

6.1 Generating the Simplified Constraint Check

In order to generate code fragments to check specific updates against a constraint, we convert the ICode representation into a structure known as a *constraint graph.* Not to be confused with the constraint graphs of Urban and Desiderio (1992), which are groupings of several constraints indicating dependencies between them, the constraint graphs of P/FDM are representations of the predicates of individual constraints. They are similar in form to the query graphs produced by the Daplex query optimiser when considering different reordering strategies for programs (Jiao 1992), and consist of a series of nodes representing the classes or scalar types involved in a chain of composed function applications, linked by the functions which navigate between them. Figure 1 shows a constraint, its ICode representation and its constraint graph. For the simple class of constraints currently implemented, constraint graphs take the form of two sub-graphs and a comparison operator:

```
cgraph(LHS_Graph, Comparison, RHS_Graph)
```

The Prolog representation of the graph given in Figure 1, then, is:

```
cgraph([class(residue, var(evar1)), function(pos), class(integer, var(evar2))],
        expression(>=, var(evar2), var(evar4)),
        [class(residue, var(evar1)), function(residue_structure),
         class(structure, var(evar3)), function(start),
                                              class(integer, var(evar4))]
)
```

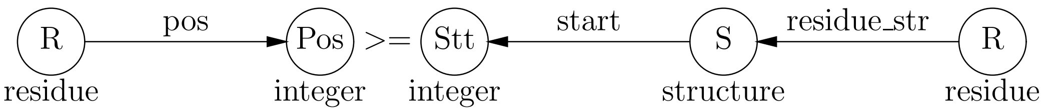
The two subgraphs are referred to as the left hand side graph (LHS) and the right hand side graph (RHS) respectively. A conjunction of predicates is represented by a `logical_and/2` term, in which each of the arguments is either another conjunction or a constraint graph.

```
      constrain each r in residue
    to have pos(r) >= start(residue_structure(r));
```

(a) *Daplex representation*

```
restrict_subquery(all(var(evar1)),
     generate(residue, var(evar1)),
     restrict(pos, [residue], [var(evar1)], var(evar2)),
generate(residue_structure, [residue], [var(evar1)], structure, var(evar3)),
     restrict(start, [structure], [var(evar3)], var(evar4)),
     expression(>=, var(evar2), var(evar4))
])
```

(b) *Intermediate code representation*



(c) *Constraint graph representation*

Figure 1 Three representations of a Daplex constraint

The update for which the fragment is to be generated is itself converted into a subgraph, that is then compared with both the constraint subgraphs, looking for matches. The subgraph templates for each update event are:

- `newentity(Class, _, _)`

  ```
  [function(KF1), class(Class, Var), function(KF2)]
  ```

- `include(Class, _, _)`

  ```
  [class(Class, Var)]
  ```

- `deletentity(Class, _)`

  ```
  [class(Class, Var)]
  ```

- `addfnval(FName, [Type | _], _)`
  `deletefnval(FName, [Type | _], _)`
  `updatefnval(FName, [Type | _], _, _)`

  ```
  [class(Type, Var1), function(FName), class(ResType, Var2)]
  ```

Each match of one of these templates for which the variables are distinct represents a potential violation of the constraint by the update, and results in the generation of a separate code fragment. This criterion on matches (i.e. that the variables be distinct) is important because it prevents us from generated redundant code fragments when the same data is referenced in more than one constraint graph. For example, although the `residue_structure` function is referenced twice in the following constraint:

```
constrain each r in residue
to have pos(r) >= start(residue_structure(r)) and
    pos(r) =< end(residue_structure(r));
```

we only need to generate one constraint checking fragment for updates to it, since the function is only ever applied to the variable `r`. However, when the variables are not the same, as in the following constraint:

```
constrain each sc in sub_component
to have disulphide(disulphide(sc));
```

then we must generate two separate fragments, one checking the case when the update is to the first application of `disulphide`, and one checking the case when the update is to the second.

When a match has been found, the graph in which it occurs is split into two further subgraphs, known as the forward graph and the reverse graph. For example, returning to our `residue` constraint, the update event `event(addfnval, start, structure)` is translated into the subgraph:

```
[class(structure, VarS), fn(start), class(integer, VarStt)]
```

and is then compared with the two subgraphs representing the constraint. In this case, there is only one possible match, with the RHS graph, and this is split at the node `class(structure, _)`, which is the point of the match, to give the forward and reverse graphs:

Forward graph:

```
[class(integer, var(evar4))]
```

Reverse graph:

```
[class(residue, var(evar1)), fn(residue_structure),
                                        class(structure, var(evar3))]
```

with `VarS = var(evar3)` and `VarStt = var(evar4)`.

We now use our three subgraphs to generate the Prolog fragment. Beginning with the two graphs involved in the match, we use the reverse graph to generate the code that instantiates the "anchor" variable (`var(evar1)/P`) by reversing it so that it begins with the variable for which the update event has supplied a value (`var(evar3)/S`):

```
[class(structure, var(evar3)), fn(residue_structure_inv),
                                        class(residue, var(evar1))]
```

(Notice that relationship functions are converted to their inverses when constraint graphs are reversed.) This graph can now be translated back into ICode format:

```
ICode1 = [generate(residue_structure_inv, [structure], [var(evar3)],
                                        residue, var(evar1))
]
```

The forward graph does not need to be reversed, as it already begins with a known variable (`var(evar4)`) and it can be converted to ICode without further modification. In our example, the forward graph consists of only a single node, and therefore produces no ICode construct. It represents simply the fact that its variable is instantiated by the arguments to the update event.

```
ICode2 = []
```

The unmatched constraint graph, LHS, can also be converted unmodified back to ICode:

```
ICode3 = [restrict(pos, [residue], [var(evar1)],
                                        var(evar2))]
```

and finally these three fragments of ICode can be glued together (in the order in which they were generated), combined with the comparison expression (negated for *for all* constraints) and translated into Prolog using the existing code generator:

```
getfnval(residue_structure_inv, [S], R),
getfnval(pos, [R], Pos),
Pos < Stt.
```

Figure 2 illustrates this process diagrammatically.

Although this constraint graph technique is apparently suitable only for use in simplifying this restricted set of constraints, in fact it is rather more general. In essence, constraint graphs can be seen as a convenient representation in which to analyse chains of function applications, and to reorder them so that they begin with variables of known value. They also have the advantage of easily distinguishing between distinct references to particular functions and redundant references, which allows us to generate only those fragments which are actually required to check the constraint and cuts out yet more redundant constraint checking.

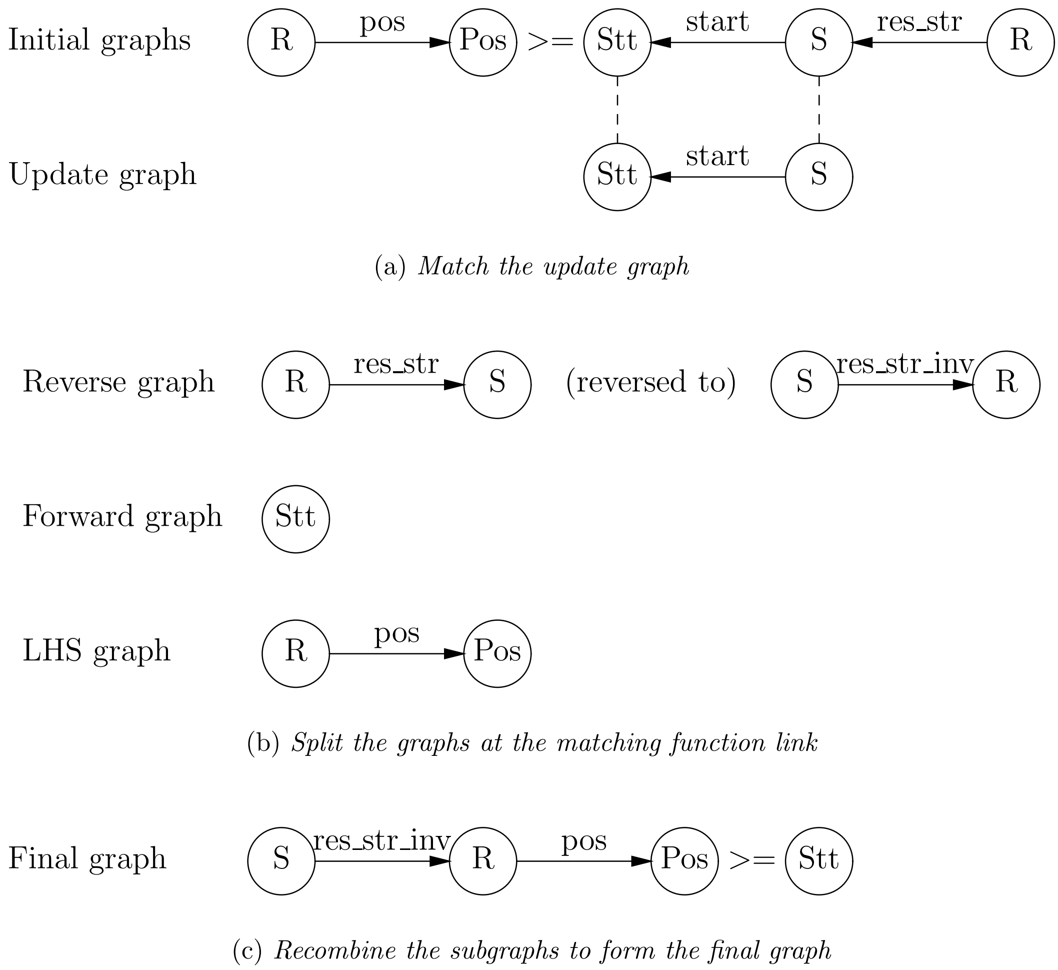


Figure 2 Reformulation of example constraint graph for code generation

6.2 Generating the Prolog Clauses

For constraints which are to be checked by standard simplified code fragments, the code generated by the above process is sufficient to express the entire constraint check, i.e. it will succeed if the constraint is violated and fail if it is not. The final step in the process is to turn the code fragment into a Prolog clause, with arguments containing the details of the update event. We also add a call to a routine to display an error message when the constraint is violated. The final clause generated to check the example `residue` constraint for the `addfnval(start(structure))` event, then, is:

```
c1_addfnval1([S], Stt) :-
     getfnval(residue_structure_inv, [S], R),
     getfnval(pos, [R], Pos),
     Pos < Stt,
     constraint_error(c1, addfnval(start, [S], Stt)).
```

For the remaining constraint types, the simplified code fragment succeeds if the update will cause a change in the number of times the constraint is satisfied, and will fail otherwise. If the fragment was generated from a potentially violating event, then this change may cause the entire constraint to be violated. The Prolog clause with represents the simplified form of the entire constraint, then, must check that the instance count will not go beyond the specified limit, and if not it must update the counter to reflect its new value. For example, given the following constraint:

```
constrain at least 5 p in protein
to have molecular_weight(p) > 2000;
```

the following Prolog clause will be generated to check the `deletefnval(molecular_weight(protein))` event:

```
c2_deletefnval1([P], MW) :-
     MW > 2000,
     get_constraint_counter(c2, CounterValue),
     (CounterValue = 5 ->
          constraint_error(c2, deletefnval(molecular_weight, [P], MW))
     ;
          decrement_constraint_counter(c2)
          fail
     ).
```

For those updates which can never violate the constraint, such as additions for *at least* constraints and deletions for *at most* constraints, we do not need to check the constraint counter's value, although we may need to adjust its value. For example, the simplified clause generated for the event:

```
addfnval(molecular_weight(protein))
```

from the above constraint is:

```
c2_addfnval1([P], MW) :-
     MW > 2000,
     increment_constraint_counter(c2),
     fail.
```

7 CONCLUSIONS

As database technology is being used to tackle more and more complex application areas, such as those found in the design and scientific fields, database administrators and users are requiring more intelligent support from the DBMS in order to be able to cope with the increasing complexity of data. Constraints, specified in a declarative fashion, offer a potentially rich source of semantic information for intelligent DBMS's, and database designers should be encouraged to include as many constraints as is possible into their schemas. However, as long as constraint maintenance imposes large overheads on the day to day interaction with databases, database designers will be reluctant to specify any but the very simplest of constraints. Our contention is, therefore, that the provision of highly efficient constraint checking mechanisms is an important initial step on the road to providing this kind of intelligent user support for databases.

The architecture and compiler described in this paper have been designed with the aim of achieving a high degree of efficiency in constraint checking. The basis of our methodology is to identify the primitive update events that can alter the truth value of a given integrity constraint considered as a Boolean function on the state of the database objects. There is a many-to-many relationship between events and constraints, in that a constraint can be affected by many types of event, and any one event may violate many constraints. This relationship must be modelled accurately in order to achieve a high degree of efficiency in constraint checking. We have described a formal systematic method for identifying the events in a rich constraint language, and have used this as the basis of our implementation. The implementation is complete, in that we can generate code to check any constraint that is expressible in the language, albeit in a less than efficient manner. However, we have also shown a systematic way to transform a certain class of constraints into the minimal code fragments that need to be executed as a check for each event. We intend to widen this class of constraints, and thus eventually achieve efficient compilation of our full constraint language without having to compromise on expressiveness. Our system architecture also has all the features required for run-time constraint checking with the minimum of overhead, whilst retaining the ability to handle different constraint checking methods, and the ability to add and delete constraints from a populated database without requiring recompilation of method or application code. Moreover, it does not depend upon any particular data model or implementation language (Embury 1993), and has been implemented in an object-oriented database (ADAM (Bassiliades 1994)) as well as in P/FDM. We believe that previous proposals and systems have paid insufficient attention to these architectural features, which has reduced their ability to do compile-time optimisation and thus achieve acceptable constraint checking performance.

The current P/FDM system now forms a suitable platform from which to explore further techniques for the simplification of a wider range of constraints. We also need to consider the problem of checking batches of updates at transaction-commit time efficiently, for which the simplification technique is not directly suitable. One possibility which we are investigating is the use of the constraint graph representations of constraints as templates, to reduce the amount of redundant data retrieval required to check constraints over multiple updates.

Table 1 Timings showing overhead of searching for constraints to check

Update type	Time required to locate relevant constraints (ms)				
	No checking	No. of irrelevant constraints			
		0	10	50	100
newentity/deletentity	198.1	229.0	228.7	229.1	229.4
addfnval/deletefnval	25.8	56.2	55.9	56.4	55.9
		Overhead per update (ms)			
newentity/deletentity	-	15.5	15.3	15.5	15.7
addfnval/deletefnval	-	15.2	15.1	15.3	15.1

ACKNOWLEDGEMENTS

Suzanne Embury is supported by the EPSRC. The authors are members of the ACT-NET "Human, Capital and Mobility" research network, supported by the Commission of the European Union.

APPENDIX 1 TIMINGS

In order to illustrate the effects of the constraint maintenance architecture described above on the performance of a real database application, we give some timings for checking constraints against a large database of three-dimensional protein structure. The database contains the structures of 142 proteins, and consists of some 300Mb of data.

We give timings to illustrate two aspects of the system: the overhead imposed by the constraint checking architecture on updates which cannot violate any of the constraints, and the improvement in performance gained when using simplified constraint code rather than checking the entire database for validity. In each case, timings are given in milliseconds (ms), rounded to the nearest 0.1ms, and (except for the unsimplified code results in table 2) are averaged over 100 iterations.

Table 1 shows the effects of having to search for constraints to check at run-time, on two updates:

```
newentity(molecule, ['156B', 'A'], Molecule),
deletentity(molecule, Molecule).
```

and

```
addfnval(residue_structure, [residue(247)], structure(4725)),
deletefnval(residue_structure, [residue(247)], structure(4725)).
```

The first of these (`newentity/deletentity`) is of average complexity and involves creating an instance of a root class, with a compound key with a single key-dependency link

Table 2 Timings showing effect of simplification on constraint checking

Constraint	Instances in class	Update checked	Time to execute constraint code (ms) Unsimplified	Simplified
C1	142	`addfnval/resolution`	360.0	0.2
		`updatefnval/resolution`	”	0.2
C2	41852	`addfnval/start`	264 694.0	31.2
		`updatefnval/start`	”	31.0
		`addfnval/end`	”	30.9
		`updatefnval/end`	”	31.0
		`updatefnval/pos`	”	6.1
		`addfnval/res_str`	”	6.1
		`updatefnval/res_str`	”	6.1
		`addfnval/res_str_inv`	”	6.0
		`updatefnval/res_str_inv`	”	6.2

(i.e. `156B` is the key of some other object). The instance is then deleted, so that the update may be repeated without creating duplicate keys in the database. This allows us to give more accurate timings averaged over several iterations. The second update illustrates a much simpler, function-based update, in which a single-valued relationship function is populated and then deleted.

The first column of table 1 gives the timings for these updates in P/FDM when the constraint maintenance subsystem is not installed. These figures give the base timings against which all the others are compared. The remaining columns give the timings for the updates in the presence of increasing numbers of constraints, none of which can be violated by the updates. The final two rows of the table give the time taken to search the metadata for relevant constraints *per update operation* (i.e. the overhead for the entire benchmark update divided by two). These results show that the overhead is low in comparison with updates of average complexity (such as `newentity/deletentity`), and that it is not affected by the number of constraints declared on other classes or functions in the database.

Table 2 gives the timings for executing the simplified and unsimplified code generated from two constraints, one simple and iterating over a small class (*C1*):

```
constrain each p in protein to have resolution(p) >= 0;
```

and the other more complex and iterating over a much larger class (*C2*):

```
constrain each r in residue to have
pos(r) >= start(residue_structure(r)) and
pos(r) =< end(residue_structure(r));
```

The second column of the table gives the number of instances in the main class in each constraint, and thus gives the number of instances over which the unsimplified constraint check must iterate. The third column gives the update for which the constraint has been simplified, and the final two columns give the times to execute the unsimplified and simplified code fragments respectively for each of these updates. The figures given here do not include the overhead required to locate the constraint code fragments. We have included these results not to show that simplified code is more efficient than unsimplified code — this is obviously true when moving from an iteration over many instances to an iteration over just one — but to show how dramatic the improvement can be for a sizeable database. These figures also demonstrate the importance of being able to identify the minimal set of potential violating events even for constraints which cannot be simplified. When constraint checking is such a time-consuming process, it is vital that it is only performed in cases where it is absolutely necessary.

APPENDIX 2 THE SYNTAX OF THE EXTENDED DAPLEX LANGUAGE

```
<command>         ::= <declarations> | <definition> | <imperative>

<declarations>    ::= create <module_status> module <mname> [<decs>]
                                [consult methods file <file_list>]
                    |  extend module <mname> [<decs>]
                                [consult methods file <file_list>]
                    |  check sybase module <mname>

<module_status>   ::= shared | private | temporary

<decs>            ::= <entity_dec> [<decs>]
                    |  <function_dec> [<decs>]
                    |  <action_dec> [<decs>]
                    |  <key_dec> [<decs>]

<entity_dec>      ::= declare <typeid> ->> <typeid>
                    |  declare <typeid> ->> entity

<function_dec>    ::= declare <fname> ( [<type_list>] ) <arrow> <typeid>
                    |  define <fname> ( [<type_list>] ) <arrow> <typeid>

<arrow>           ::= -> | ->>

<action_dec>      ::= define <aname> ( <type_list> )

<type_list>       ::= <typeid> [, <type_list>] | set of <typeid>

<file_list>       ::= <filename> [, <file_list>]

<key_dec>         ::= key_of <typeid> is <key_components>
```

```
<key_components>  ::= <key_component> [, <key_components>]

<key_component>   ::= key_of ( <fname> ) | <fname>

<definition>      ::= <program_def> | <function_def> | <action_def>

<program_def>     ::= program <pname> is <imperative>

<function_def>    ::= define <fname> ( [<define_args>] ) ->
                                   <typeid> in <mname> <singleton>
                     |  define <fname> ( [<define_args>] ) ->>
                                   <typeid> in <mname> <set_expr>

<action_def>      ::= define <aname> ( [<define_args>] ) in <mname>
                                [ensuring <predicate>] <imperative>

<define_args>     ::= [set of] <varid> in <typeid> [, <define_args>]

<imperative>      ::= [<loops>] <action_part>

<loops>           ::= <for_loop> [<loops>]

<for_loop>        ::= for each <named_set_expr>
                     |  for <limited_set_expr>
                     |  for <singlevar>

<action_part>     ::= <action_statement> [, <action_part>]

<action_statement>::= <print_statement> | <update_statement> | <action>

<print_statement> ::= print ( <expr_list> )

<update_statement>::= let <svfncall> = <singleton>
                     |  let <mvfncall> = <expr>
                     |  include <set_expr> into <fcall>
                     |  exclude <set_expr> from <fcall>
                     |  delete <expr> |
                     |  create a new <typeid> with key = ( <key_values> )
                     |  create a new <var> in <typeid> with key = ( <key_values> )

<fcall>           ::= <svfncall> | <mvfncall> | <typeid>

<key_values>      ::= <key_value> [, <key_values>]

<key_value>       ::= key_of <singleton> | <singleton>
```

```
<action>           ::= <aname> ( <expr_list> ) | <aname> ( <set_expr> )

<expr_list>        ::= <expr> [, <expr_list>]

<expr>             ::= <set_expr> | <singleton>

<named_set_expr>   ::= <varid> in <set> [such that <predicate>] [as <typeid>]

<set_expr>         ::= <set> [as <typeid>]

<set>              ::= <typeid>
                    |  <mvfncall>
                    |  if <predicate> then <set> else <set>
                    |  ( <set_expr> <set_operation> <set_expr> )
                    |  { <explicit_component> }
                    |  <varid> in <set_expr> such that <predicate>

<set_operation>    ::= union | intersection | difference

<explicit_component>
                   ::= <singleton> [to <singleton>] [, <explicit_component>]

<singleton_list>   ::= <singleton> [, <singleton_list>]

<singleton>        ::= <const>
                    |  <varid> [as <typeid>]
                    |  <svfncall>
                    |  <singlevar>
                    |  <arith_expr>
                    |  <aggcall>
                    |  ( <singleton> )
                    |  <conditional_expression>

<singlevar>        ::= the <named_set_expr>

<limited_set_expr> ::= any <named_set_expr>
                    |  any <singleton> <named_set_expr>

<arith_expr>       ::= [+ | -] <arith_term>

<arith_term>       ::= <arith_fac>
                    |  <arith_term> + <arith_fac>
                    |  <arith_term> - <arith_fac>

<arith_fac>        ::= <singleton>
                    |  <arith_fac> * <singleton>
```

```
                     |  <arith_fac> / <singleton>

<aggcall>           ::= <summing_op> ( over <named_set_expr> of <singleton> )
                     |  <aggregate_op> ( <set_expr> )

<summing_op>        ::= total | average

<aggregate_op>      ::= maximum | minimum | count

<conditional_expression>
                    ::= if <predicate> then <singleton> else <singleton>

<predicate>         ::= <bool_term> | <predicate> or <bool_term>

<bool_term>         ::= <bool_fac> | <bool_term> and <bool_fac>

<bool_fac>          ::= [not] <bool_prim>

<bool_prim>         ::= <comparison>
                     |  <quantified_expr>
                     |  <set_membership>
                     |  <subclass_membership>
                     |  ( <predicate> )

<comparison>        ::= <arith_expr> <comparison_operator> <arith_expr>

<comparison_operator>
                    ::= = | < | > | <> | =< | >=

<quantified_expr>   ::= <quantifier> <named_set_expr> <to_have> <predicate>

<quantifier>        ::= all | each | some | any | no
                     |  exactly <const> | at <range_quantifier> <const>

<range_quantifier>::= least | most

<to_have>           ::= has | have | to have

<set_membership>    ::= <singleton> in <set_expr>

<subclass_membership>
                    ::= <singleton> is a <typeid>

<mvfncall>          ::= <fname> ( [<expr_list>] )

<svfncall>          ::= <fname> ( [<singleton_list>] )
```

where

`<aname>` is the name of an action,
`<const>` is an integer, float, string or boolean constant,
`<filename>` is the name of a file,
`<fname>` is the name of a function,
`<mname>` is the name of a module,
`<pname>` is the name of a program,
`<scalar>` is a scalar literal,
`<typeid>` is the name of a type, and
`<varid>` is the name of a variable.

REFERENCES

Bassiliades, N. (1993) Constraint Description in ADAM, MSc. Thesis, Dept. of Computing Science, University of Aberdeen, Aberdeen, Scotland, AB9 2UE.

Bassiliades, N. and Gray, P.M.D. (1994) CoLan: a Functional Constraint Language and Its Implementation. *Data and Knowledge Engineering*, **14**, 203–249.

Bry, F., Decker, H. and Manthey, R. (1988) A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases, in *Advances in Database Technology - EDBT '88*, Schmidt, J.W., Ceri, S. and Missikoff, M. (eds.). Springer Verlag, 488–505.

Bry, F., Manthey, R. and Martens, B. (1992) Integrity Verification in Knowledge Bases, in *Logic Programming: Proc. of the First and Second Russian Conference on Logic Programming*, Voronkov, A. (ed.). Springer-Verlag, 114–139.

Ceri, S. and Widom, J. (1990) Deriving Production Rules for Constraint Maintenance, in *Proceedings of the 16th VLDB Conference*, Sacks-Davis, R., McLeod, D. and Schek, H. (eds.). Brisbane, Morgan Kaufmann Publishers, Inc., 566–577.

Díaz, O. (1992) Deriving Rules for Constraint Maintenance in an Object-Oriented Database, in *DEXA 92 Conference*, Toja, A.M. and Ramos, I. (eds.). Springer-Verlag, 332–337.

Díaz, O. and Embury, S.M. (1992) Generating Active Rules from High-Level Specifications, in *Advanced Database Systems: Proceedings of 10th British National Conference on Databases (LNCS 618)*, Gray, P.M.D. and Lucas, R.J. (eds.), Aberdeen, Scotland. Springer Verlag, 227–243.

Embury, .S.M. and Gray, P.M.D. (1995) The Declarative Expression of Semantic Integrity in a Database of Protein Structure, in *Proceedings of Basque International Workshop on Information Technology*, Illarramendi, A. and Díaz, O. (eds.), San Sebastian, Spain. IEEE Society Press, 216–224.

Embury, S.M. (1994) Constraint-Based Updates in a Functional Data Model Database. PhD Thesis, University of Aberdeen, King's College, Aberdeen, Scotland.

Embury, S.M., Gray, P.M.D. and Bassiliades, N.D. (1993) Constraint Maintenance Using Generated Methods in the P/FDM Object-Oriented Database, in *Proc. of 1st Int. Workshop on Rules in Database Systems (RIDS '93)*, Paton, N.W. and Williams, M.H. (eds.) Workshops in Computing, Edinburgh. Springer-Verlag, 364–381.

Erwig, M. and Lipeck, U.W. (1991) A Functional DBPL Revealing High Level Optimizations, in *Proceedings of the 3rd International Workshop on Database Programming Languages: Bulk Types and Persistent Data*, Kanellakis, P. and Schmidt, J.W. (eds.), Nafplion, Greece. Morgan Kaufmann Publishers, Inc., 306–321.

Fraternali, P. and Paraboschi, S. (1993) A Review of Repair Techniques for Integrity Maintenance, in *Proc. of 1st Int. Workshop on Rules in Database Systems (RIDS '93)*, Paton, N.W. and Williams, M.H. (eds.) Workshops in Computing, Edinburgh. Springer-Verlag, 332–346.

Gaasterland, T. and Lobo, J. (1994) Qualified Answers that Reflect User Needs and Preferences, in *Proceedings of the 20th VLDB Conference*, Bocca, J., Jarke, M. and Zaniolo,C. (eds.), Santiago de Chile. Morgan Kaufmann Publishers, Inc., 309–320.

Gehani, N.H. and Jagadish, H.V. (1991) Ode as an Active Database: Constraints and Triggers, in *Proceedings of the 17th VLDB Conference*, Lohman, G., Sernadas, A. and Camps, R. (eds.), Barcelona. Morgan Kaufmann Publishers, Inc., 327–336.

Gray, P.M.D., Kulkarni, K.G. and Paton, N.W. (1992) *Object-Oriented Databases: a Semantic Data Model Approach*. Prentice Hall Series in Computer Science. Prentice Hall International Ltd.

Jagadish, H.V. and Qian, X. Integrity Maintenance in an Object-Oriented Database, in *Proceedings of the 18th VLDB Conference*, Yuan, L.-Y., (ed.), Vancouver. Morgan Kaufmann Publishers, Inc., 469–480.

Jiao, Z. (1992) *Optimisation Studies in a Prolog Object-Oriented Database*. PhD Thesis, University of Aberdeen, Aberdeen, Scotland.

J.J. King. (1984) *Query Optimisation by Semantic Reasoning*. UMI Research Press.

Landin, P.J. (1966) A Lambda Calculus Approach. in *Advances in Programming and Non-Numerical Computation*, Fox, L. (ed.), Symposium Publications Division, Pergamon Press, chapter 5, 97–141.

Nicolas, J.-M. (1992) Logic for Improving Integrity Checking in Relational Databases. *Acta Informatica*, **18**, 227–253.

Paton, N.W. and Gray, P.M.D. (1990) Optimising and Executing Daplex Queries Using Prolog. *The Computer Journal*, **33**, 547–555.

Poulovassilis, A. and Small, C. (1991) A Functional Programming Approach to Deductive Databases. in *Proceedings of the 17th VLDB Conference*, Lohman, G., Sernadas, A. and Camps, R. (eds.), Barcelona. Morgan Kaufmann Publishers, Inc., 491–500.

Qian, X. (1993) The Deductive Synthesis of Database Transactions. *ACM Transactions on Database Systems*, **18**(4), 626–677. 1993.

Reddi, S. (1993) Integrity Constraint Enforcement in the Functional Database Language PFL, in *Proceedings of the 11th British National Conference on Databases*, Worboys, M. and Grundy, A.F. (eds.), Keele, U.K. Springer-Verlag, 238–257.

Shipman, D.W. (1981) The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, **6**(1), 140–173.

Urban, S.D. (1989) ALICE: an Assertion Language for Integrity Constraint Expression. in *Proceedings of Conference on Computer Software Applications*.

Urban, S.D. and Desiderio, M. (1992) CONTEXT: a CONstraint EXplanation Tool. *Data and Knowledge Engineering*, **8**, 153–183.

APPENDIX 3 BIOGRAPHY

Suzanne Embury is a research fellow in the Department of Computing Science at Aberdeen University, working on a project to provide improved and more declarative language support for updates in large scientific databases. She received a B.Sc. in Computer Science from the University of Kent at Canterbury, in 1990, and her Ph.D. from the University of Aberdeen, in 1994, on the role of semantic constraint information and non-determinism in updating functional data model databases. Her current interests include compiler design for high-level data manipulation languages, and the use of constraint logic programming techniques within database management systems, as a substrate for declarative update languages and in the generation of transaction repair plans.

Peter Gray is a professor in the Department of Computing Science at Aberdeen University, where he has been since 1968. He originally worked on the ABSYS system where he implemented persistent storage of list structured objects in a very early equational or constraint logic programming system. This lead to an interest in logic programming and automatic program generation applied to scientific database systems. Currently, he is leading work which applies these ideas to protein structure design and engineering design, using a large object database based on a semantic data model. He has written and co-authored several books on this, including "Logic, Algebra and Databases (1985)". He received his B.A. in Physics from Cambridge (1961) and D.Phil. in elementary particle physics and computing from Oxford (1965). He is a fellow of the BCS, and European Chair for VLDB (1995).