POL: Persistent Objects with Logic

Paul Adams
Marvin Solomon

Technical Report #1158

June 1993

POL: Persistent Objects with Logic[*]

Paul Adams
adams@cs.wisc.edu
and
Marvin Solomon
solomon@cs.wisc.edu

POL is a combination of elements from three domains: object-oriented programming, logic programming, and database. POL integrates these domains using a shared data model and a close coupling of an existing object-oriented language, C++, with a logic-programming language called Congress. We describe the components of POL, with particular emphasis on the interaction between the object-oriented features of C++ and the logic-based features of Congress. We then illustrate the power of POL with three sample applications: a C++ program database, a bibliographic database, and a Unix-compatible file system for software configuration.

1. Introduction

POL (Persistent Objects with Logic) is a mixture of three styles of programming languages: object-oriented, logic-based, and persistent. Each style has features that make solving certain problems easier: Object-oriented languages encapsulate state and behavior and support extension by inheritance; logic programming languages allow programmers to concentrate on describing *what* a solution is rather than *how* to find it; persistent programming languages relieve the programmer of the burden of saving and restoring data. By combining features from all three domains, POL provides an environment in which application programmers can take advantage of the particular style that best suits the problem at hand.

POL derives its object-oriented features from C++ [9] (§2.2), persistence from the Exodus database toolkit [5] (§2.3), and logic-based features from Congress—a new language derived from Prolog [6] and LOGIN [1] (§2.4). POL integrates these components with a common data model and a close coupling of Congress with C++. (Although we make specific references to C++, any object-oriented language that has the essential features of encapsulation and inheritance could be used.)

The data model provides a common abstraction for the logic-based and object-oriented features of POL. From the perspective of logic programming, database objects are interpreted declaratively and are used to construct programs and queries. From the perspective of object-oriented programming, database objects are class instances sharing a common interface.

[*]This work was supported in part by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937).

Authors' address: Computer Sciences Department, University of Wisconsin—Madison, 1210 W. Dayton St., Madison, WI 53706.
(608) 263-2844 or (608) 262-5776.

The coupling between C++ and Congress is a two-way embedding: Each language appears to be an embedded sublanguage of the other. Each language retains its own style; the embedding does not alter the syntax or semantics of either language. Moreover, new data types that encompass both a procedural and declarative aspect can be defined and used in both C++ and Congress as long as they conform to the abstract data model.

The object-oriented features of C++ play an important role in making the embedding possible: Abstract classes are used to represent Congress data and procedures and the dynamic state of the Congress interpreter is encapsulated in a class. These abstractions provide a tighter degree of integration than is possible with a simple embedded interpreter and make the access to the other language more natural in that many of the differences are hidden by the abstractions.

Each database element has a unique identity. Object identifiers are first-class values that may be embedded in objects to create complex data structures. Manipulating the identity of objects is natural in a language with pointer types such as C++. To achieve a similar effect in Congress, we have extended the "value-based" semantics of Prolog and provided Congress the ability to access the identity of database objects. For example, Congress has an assignment operator that allows "database facts" to be updated in place. Thus updates to the database can be made either from C++ or Congress, whichever is more convenient.

The remainder of this paper is organized as follows. Section 2 describes the data model and each component of POL. Section 3 contains more detail on the embedding of the logic and object-oriented languages. Section 4 illustrates the utility of POL with three applications: a lint-like tool for C++, a document and bibliography formatting system, and an attributed filesystem. Related work is discussed in Section 5. We conclude with a report on the current status of POL and our plans for the future.

2. Components of POL

This section describes the major components of POL and the relationships between them. We start with the data model and then present in turn the object-oriented, persistent, and logic based facets.

2.1. Term Space

The common basis for all of POL is a global database called the *term space*, which is a directed graph with labelled nodes and arcs. The label associated with a node is called its *functor*[1] and the label associated with an arc is called its *selector*. No two arcs leaving the same node may have the same selector. A *term* is the subgraph of the term space reachable from a node, called the *root* of the term. We occasionally identify a term with its root node, when the meaning is clear from context. For example, the "functor of a term" means the functor of its root node.

The term space is "identity-based": Two nodes with identical contents are nonetheless considered to be distinct. Nodes are explicitly created, and updates to a node do not change the node's identity. In this way POL differs from "value-based" Prolog and relational databases, and more closely resembles so-called "object-oriented" databases. The term space is partitioned into *program* and *data* subspaces, with no arcs between them. The program subspace is read-only from Congress.

All terms in the term space are persistent. POL assists with managing changes to the term space by supporting multiple versions of the term space called *worlds* using an algorithm devised by Driscol *et. al.* [7]. POL has operations to save the current term space as a world, and to reset its state to any previously saved world. A save operation does not copy the entire term space, but only an amount of data proportional to the changes made since the previous save.

[1]This unfortunate choice of terminology is inherited from Prolog.

2.2. C++

C++ is a strongly typed object-oriented language derived from C. C++ classes encapsulate both data and operations on that data. C++ supports multiple inheritance and information hiding via explicit public/private declarations. Subclasses can override methods of their super-class as well as add new data fields and operations. We assume the reader has a basic reading knowledge of C++. Our examples will avoid the more exotic features of the language. Excellent texts on C++ have been written by Stroustrup [25] and Lippman [17].

C++ classes are used in POL to provide a concrete realization of term space nodes and arcs. All classes are derived from the abstract base class `Term` which represents a node and its outgoing arcs. Subclasses are `Atomic` and `InternalNode`. Atomic nodes have no outgoing arcs. They are further classified according to the data types of their functors: integers, real numbers, printable strings, byte strings (arbitrary binary data) or "variables." (Variables are explained in Section 2.4.) An internal node contains a functor (which must be a printable string) and a table of references to other nodes indexed by distinct printable strings. Internal nodes are similar to C structs, Pascal records, SNOBOL tables, CLOS objects, and AWK associative arrays. Unlike structs or records, the number and names of "fields" may vary dynamically and their contents are restricted to be non-null pointers to nodes. C++ subclass derivation is used to add additional behavior and restrictions to classes of internal nodes. We shall return to this point in Sections 3 and 4.

2.3. Exodus

Exodus [5] is a toolkit for creating custom database systems. POL uses two components of Exodus, a low-level storage subsystem called the *Exodus Storage Manager* and a persistent dialect of C++ called *E*. The Exodus Storage Manager provides efficient access to arbitrary-sized untyped persistent arrays of bytes called "storage objects," which are identified by unique object identifiers ("OIDs"). The Storage Manager supports concurrency control with two-phase locking, and serializable transactions with full recovery from hardware and software failures. The E programming language [22] is an extension of C++ that supports persistent data—data that retains its state between runs of a program. E syntax extends C++ with a "db" version of each primitive type and type constructor (`dbint, dbclass { ... }`, etc.). Instances of db types can be either transient or persistent depending on how they are created. Pointers to persistent dbtypes are represented by OIDs. When a db pointer is dereferenced, the corresponding object is fetched from disk and the pointer is "swizzled" to a memory address. The object is flushed back to disk and pointers to it are unswizzled at the end of a transaction. POL implements the term space with persistent data structures.

Throughout this paper, all references to the C++ programming language should be understood as referring to the E dialect of C++.

2.4. Congress

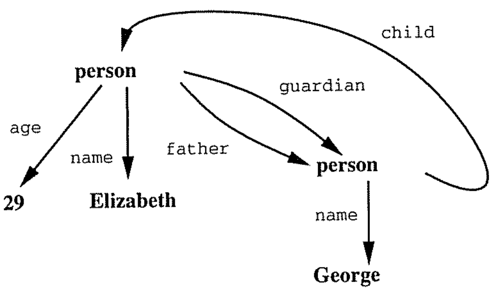
The heart of POL is a new language called *Congress*. Congress may be described as a logic programming language, a deductive database query language, an embedded query language, or a library of classes for convenient database access, depending on one's point of view. Since Congress is implemented as a library of classes, any C++ program can use Congress as a "higher level" alternative to or enhancement of the raw C++ term interface.

As a logic-programming language, Congress is a dialect of LOGIN [1], an extension of Prolog supporting cyclic terms. It provides transparent persistence, and extends the valued-based semantics of Prolog with the ability to manipulate the identity of (data) nodes.

The following paragraphs briefly describe the syntax and semantics of Congress and compare it with Prolog; Section 3 discusses features, such as update semantics, that derive from Congress' embedding in C++. The reader who is familiar with logic programming may skim this section.

Congress programs are built from terms in the POL term space. A *program* is a set of *procedures*, a procedure is a sequence of *clauses*, and a clause is a sequence of terms. A clause consists of a single term called its *head* and a sequence of zero or more additional terms called its *body*. The *predicate* of a clause is the functor of the root node of its head term. A procedure is a sequence of clauses with a common predicate, referred to as the *name* of the procedure. A program is a set of procedures with distinct names.

Congress has a character-string textual representation that may be used to enter or print programs or fragments of programs, or to enter queries from the keyboard. A term t may be denoted `f(`s_1`=>`t_1`, ··· ,`s_n`=>`t_n`)`, where `f` is its functor, $s_1, \cdots, s_n$ are the selectors of the arcs with `f` as their tail, and $t_1, \cdots, t_n$ are textual representations of the terms at the heads of the corresponding arcs. A variable is denoted "`@`." A *tag* (an alphanumeric string starting with a capital letter) is used to indicate shared subtrees or cycles. For example, the term



may be denoted[2]

```
F:person(
    name=>  "Elizabeth",
    age=>     29,
    father=>  G:person( name => "George", child => F ),
    guardian=> G
).
```

A clause with head t_0 and body $t_1, \ldots, t_n$ is denoted "t_0 `:-` $t_1, \ldots, t_n$." The textual representation also supports infix representation for common functors such as + and * and Prolog notation for lists. For example, the expression `[a, b | Tail]` denotes the same term as the expression `cons(car => a, cdr => cons(car => b, cdr => Tail))`. A missing selector implies an edge labelled with an integer and "`:@`" following a tag may be omitted. For example, `f(a,X)` represents the same term as `f(1=>a, 2=>X:@)`.

With these abbreviations, the set of textual representations constitutes a superset of the syntax of Prolog. Congress is like Lisp—and different from most Algol-like languages—in that the abstract syntax (the term-space data structure) is considered the "true" representation and the concrete or "surface" syntax (the textual representation) is treated as a crutch for entering program fragments or displaying them. This distinction is particularly important in the context of the embedding of Congress in C++, where it is much more convenient to manipulate Congress terms as data structures than character strings. The mapping between terms and expressions is not one-to-one in either direction. On one hand, a term may have many textual representations differing in the order of arguments, the spelling of tags, or the unrolling of cycles. On the other hand, one textual expression may describe two isomorphic

[2]A functor that contains non-alphanumeric characters or starts with an upper-case letter must be quoted.

terms with distinct identities.

Congress extends Prolog in two important ways. First, the successors of a node are indicated by keyword rather than positional notation. This extension helps avoid programming errors.[3] For example, the Congress expression `employee(age=>25,salary=>30)` is more readable than the corresponding Prolog expression `employee(25,30)`. Second, while Prolog terms are trees (except for identification of multiple occurrences of the same variable), Congress allows arbitrary graphs, including cycles. Variables serve two purposes in Prolog: They represent "wild cards" for pattern matching and they indicate sharing. Congress uses the functor "@" for a wild card, while sharing is represented directly in the data structure and indicated textually with tags.

The operational behavior of Congress is defined by the same recursive backtracking search as in Prolog. A *goal* (or "query") consists of a term. It is *called* (*evaluated, proved*) by searching for a clause t_0 :- t_1, ..., t_n whose head t_0 "matches" the goal. If no such clause is found, the call *fails*. Otherwise the first matching clause is chosen and each of the terms t_1, ..., t_n is called in turn. When a call fails, the interpreter backs up by undoing all of its actions since the last "choice point" (the point at which a clause was chosen) and chooses another clause. The process continues either until all goals and subgoals have been successfully called, in which case the original call *succeeds,* or until all alternatives have been exhausted, in which case it *fails.*

The heart of this process is the definition of "matching" between terms, called *unification.*[4] Congress uses a variant of unification that supports cyclic terms [1]. The goal of unification is to determine if two terms are isomorphic, or can be made isomorphic by substituting terms for variables. Two terms unify if their roots match (have the same functor) and corresponding successors (recursively) unify. That is, if both roots have arcs with the same selector leaving them, the nodes reached by these arcs must also unify. As mentioned in §2.2, some nodes are designated as *variables;* a variable matches any node. A side effect of a successful unification is an equivalence relation that records which nodes were matched. Missing selectors do not prevent unification. If a selector appears in one term but not the other, an edge to a new *variable* node is added an labeled with the missing selector. For example the terms t_1 = `f(a=>g,b=>@)` and t_2 = `f(b=>h,c=>i)` unify, yielding the equivalence relation {{f_1, f_2}, {g, $@_2$}, {$@_0$, h}, {$@_1$, i}}, where f_1 and f_2 are the roots of t_1 and t_2, $@_0$ is the explicit variable node in t_1, and $@_1$ and $@_2$ are new variable nodes added to t_1 and t_2, respectively.

Computation takes place in a non-persistent *extended term space*, which contains copies of terms from the persistent term space as well as an equivalence relation representing the result of unifications. The evaluation of a call adds a *copy* of the matching clause to this extended term space and extends the equivalence relation with the result of unifying its head with the query.[5] The terms of the body are called in this modified space. If an equivalence class in the extended term space contains exactly one data node, its *identity* is be the object identifier of that data node. Classes that have no data nodes or more than one data node have no identity. (This concept is used to define the assignment operator in Section 3.4.)

3. Embedding

This section describes the bridge between C++ and Congress. Each language can be viewed as an embedded sublanguage of the other. Section 3.1 describes C++ access to the data and control structures of Congress. Access

[3]It also has a rather subtle effect on the definition of unification. See the LOGIN paper [1] for details.

[4]Background material on unification can be found in many logic programming texts and in an excellent survey by Knight [15].

[5]Copies are used to preserve the meaning of clauses as universally quantified formulæ. This copying is called "renaming the variables apart" in logic-programming literature.

to the data and control structures of C++ from Congress is described in Sections 3.2 and 3.3, respectively. Section 3.4 investigates updates from Congress in more detail.

3.1. Embedding Congress in C++

The embedding of Congress in C++ is straightforward: All of the major data structures of Congress are C++ classes, and the Congress interpreter and its control state are encapsulated by a class. Since Congress terms, clauses, procedures and programs are class instances, they can be accessed directly from C++, independently from the Congress interpreter. For example, the (abstract) class `Term`, which implements the nodes and arcs of the term space, has the interface shown in Figure 1. (All figures appear in §3.5 starting on Page 9. Some details are omitted and only public member functions are shown.) The first three methods are for "direct manipulation" of terms by C++ programs, such as the expression-language parser and an interactive browser/editor. The next three methods are used by both C++ programs and the Congress interpreter to access the state of a term. The last three methods support updates. They are discussed in Section 3.3.1.

During the evaluation of a query, the interpreter makes copies of the terms appearing in the program and records the results of unifications as an equivalence relation among the nodes of the copies (see §2.4). These copies are instances of class `ExtendedTerm` (Figure 2). The first three access functions are similar to those of `Term`. `Union` is used by the unification algorithm to record that two extended-term nodes have been matched, while `Deunion` is used while backtracking to undo bindings. `Find` returns a representative member of the equivalence class containing `this ExtendedTerm`; `t1` and `t2` are identified by the current equivalence relation if and only if `t1.Find()==t2.Find()`. An equivalence class may contain variables and non-variable nodes with equal functors. `Realize` generates a new transient term by copying the node and all nodes reachable from it, choosing from each equivalence class a non-variable node if possible. `Identity` returns the identity of this equivalence class if any (as defined in §2.4).

The volatile state of the Congress interpreter, including storage allocation and backtracking information, is encapsulated in an instance of class `Context` (Figure 3). To invoke the interpreter, a C++ program creates a new instance from a query and a program. Since each invocation of the interpreter is a separate object, it is possible for multiple invocations to exist concurrently. It is even possible for the interpreter to invoke itself recursively. `Eval()` runs the interpreter. If the top-level call succeeds, `Eval()` returns `true` and `PrintResult()` or `Result()` can be used to access any bindings that were created. Subsequent calls to `Eval()` attempt to satisfy the query in other ways. For example, given the program

```
p(X) :- q(X).
p(a).
q(b).
```

and the query `p(Y)`, `Eval()` will succeed twice and fail on the third call. `PrintResult()` will print `p(b)` after the first call, and `p(a)` after the second.

`Result()` materializes a copy of the query after applying all bindings generated by the call to `Eval()`. For example, a C++ programmer could use the code in Figure 4 to find all the answers to a query and collect those that passed a filter into a list.

3.2. Embedding C++ in Congress

From the Congress programmer's perspective two features are needed for embedded access to C++: the ability to access C++ class instances and the ability to call C++ procedures and methods. With multiple inheritance, any C++ class can be made a subclass of `Term`, thereby making its instances available for use in Congress programs. Access to C++ functions is provided through a feature called an *external predicate* (EP), which is a pair of C++

procedures that behave similarly to a Congress procedure (see Section 3.3).

An instance of any descendant of class `Term` can be accessed by a Congress program. Depending on the application, such classes can either hide all of their internal structure or present some of it as subterms. The simplest approach is to make an object appear to Congress as a leaf term. This approach is useful for representing non-term data such as binary images. For example, a class implementing Unix-like files might be defined as in Figure 5. A Congress program can access the internal structure of these "opaque" objects only by calling external predicates.

In other cases, it may be appropriate to expose some of structure of an object as edges pointing to other objects. By overriding the default edge methods, a subclass can define what its internal data looks like as a term and control the ability to modify its data. For example, suppose a Unix directory object is implemented as a hash table. A definition that makes it conform to the `Term` interface is shown in Figure 6.

3.3. External Predicates

Although a Congress procedure is represented as a list of clauses, each of which is a list of terms, the Congress interpreter only requires that it conform to the abstract interface `Procedure`, shown in Figure 7. When the interpreter needs to evaluate a query term `Q`, it looks up `Q`'s functor in a table of procedures associated with the program to find the corresponding procedure and invokes its `Call` method, passing `Q` as the first argument. The `Call` method for Congress procedures attempts to unify the query with the head of each of its clauses. If successful, it returns the corresponding body in the result parameter `continuation`. The value/result parameter `workspace` is nil on the initial call. The value returned in `workspace` by each call is passed back in the next call. A Congress procedure uses `workspace` to keep track of its place in its list of clauses. When no more clauses will unify with the query, `Call` returns `false`. The procedure `Backtracks()` indicates whether this procedure can backtrack. For Congress procedures, it returns `true` when there is more than one clause. It is used by the interpreter for important optimizations that can only apply to procedures that do not backtrack.

Class `Procedure` has two subclasses: the class `CongressProcedure` just described and `ExternalProcedure`. An `ExternalProcedure` contains pointers to a pair of C++ functions that implement `Call()` and `Clean()`. A programmer who needs to perform a function that is more conveniently implemented in C++ than in Congress need only write a pair of procedures with the appropriate interface and install them as an external predicate in the current Congress program. An EP that needs to maintain state for backtracking can allocate a block of storage and return a pointer to it in the `workspace` parameter. When a procedure that can backtrack returns `false` (indicating that no more answers are available for the current argument), the interpreter calls the `Clean` function so that the EP can clean the workspace and delete it.[6]

Construction of external predicates is illustrated by a database of documents. Deriving class `Document` from `Term` allows a Congress program to manipulate documents as Congress terms. If, moreover, the `Document` class is implemented in a manner similar to the Unix directory example above, attributes such as title, list of authors, publisher, etc. can be manipulated as subterms. A Congress procedure to check whether a particular person is an author of the document with a given title might be written as

```
is_author(T,A)  :- document(title=>T, authors=>L),
                   member(A, L).
```

While simple and easy to understand, this procedure is inefficient. It exhaustively searches all documents, testing each one to see if the supplied title and author match. The *cut* predicate could be used to halt the search once a

[6]The interpreter may also call `Clean` in certain other situations when it knows no more calls will be made.

matching title is found, but the predicate may still search the entire database in the worst case. Suppose there is a B-tree index mapping titles to document objects. Assume that the B-tree object exports a procedure `Lookup(char *title)`, which returns an `Iterator` object that iterates through all `Document` objects with a matching title. If `title` is `ALL_DOCS`, `Lookup` returns an iterator that iterates through all `Document` objects. We can add an external predicate `doc_with_title` by writing the procedures shown in Figure 8 and installing them with the statement

```
InstallEP("doc_with_title", doc_title_call, doc_title_clean, true);
```

(the last argument indicates that this EP may backtrack). The Congress procedure `is_author` becomes

```
is_author(T,A) :- doc_with_title(document(title=>T, authors=>L)),
                  member(A, L).
```

Note how `doc_title_call` uses the `unify` method of the `Context` object passed to it to compare the remaining attributes in `arg` to each candidate object returned by the index. If `arg` contains any variables, they will be be bound as a side-effect of the unification.

The new version of `is_author` still has the property that if the first argument is an unbound variable, it will find any document by the named author, but if a specific title is supplied, it will use the index to avoid considering irrelevant documents. A slightly more complicated version could take advantage of an index on author names as well.

External predicates have proven to be extremely useful. They are used to implement all of the built-in "non-logical" predicates usually found in Prolog implementations, such as arithmetic operations, `forall`, and `print`, as well as database operations and other functions that are awkward or impossible to implement directly in Congress. Examples of the latter sort of operation are file system access and invocation of other programs. The Congress interpreter is itself an external predicate so Congress programs can invoke the interpreter recursively. Implementing built-in procedures separately from the interpreter has two advantages: Different algorithms can be easily prototyped and application-specific code and data structures can be used.

3.4. Updates

A clean semantics for update operations appears to be difficult to specify for database query languages in general and for logic-based languages in particular. In Prolog, updates are performed by the built-in predicates *assert* and *retract*, which add and remove clauses from the current program. Since Prolog does not distinguish program from data (database facts are simply program clauses with empty bodies and no variables), these operations also serve to update the "database." Congress makes a clearer distinction between program and data. Although both the program and data spaces are built from the same kinds of objects (`Term` and its subclasses), they are kept separate. To support the identity-based semantics for the data space, it provides operations operations for creating and destroying data objects, and for updating an object in place.

To support the backtracking search used by Congress, each node has both an update method and an un-update method (Figure 1). The update method of a term overwrites its state with values taken from the `new_value` argument (its functor and outgoing edges), saving enough information in the `undo_record` to allow the update to be undone.

POL contains three EPs for modifying the data portion of the term space, `create`, `destroy`, and `:=` (assignment). The EP `create(term=>T)` is similar to Prolog's `assert`; it creates a copy of `T` and adds it to the current program as a new data fact. The inverse of `create`, `destroy(term=>T)`, is similar to Prolog's `retract`. It removes the data term identified by `T` from the current program and deletes its nodes from the data term space. Assignment (`:=`) is similar to assignment in an imperative language. It invokes the `Update` method on the identity (§2.4) of its left-hand side extended term with a copy of its right-hand side. (The assignment operation fails if the left-hand side extended term does not have a defined identity.) In both cases, the copy is created by traversing the extended term, replacing all program nodes with new nodes allocate in data space. Using the identity of the left-hand side is necessary to ensure that a clause such as

```
set_salary(Name, Amt) :- employee(name=>Name, salary=>Sal), Sal := Amt.
```

updates the data term *matched* by the first term in the body, rather than updating the (program) term itself. The copy operation is needed so that the call

```
?- set_Salary("Solomon", 120000).
```

changes the Employee record to point to a *copy* of the atomic node `120000` contained in the query, rather than the query itself.

Just as unifications are undone during backtracking, assignment saves the previous state of a node so that, if backtracking occurs, any changes are undone. Changes made by assignments can be frozen by preventing backtracking, either by accepting the result at the top level call of the interpreter or by use of the cut predicate [27]. Because assignment changes the (data) term space immediately, the semantics of a program that contains an assignment depends on the depth first search order followed by the interpreter. We are investigating a semantics that is more "denotational" and less bound to implementation details through introduction of an explicit "sequential operator" similar to that of Warren and Manchanda [18] or by delaying the visibility of updates using the versioning mechanism of POL.

3.5. Figures

```
dbclass Term {
public:
  // Construction
    Term(char* functor);                // create a term with no outgoing edges
    virtual Boolean AddEdge(char* selector, Term* child)
    virtual Boolean RemoveEdge(char* selector)

  // Access
    virtual char* Functor();
    virtual void Selectors(StringSet &label_set);
    virtual Term* Edge(char* selector); // traverse an edge

  // Updates
    virtual Boolean IsData();
    virtual Boolean Update(Term* new_value, TermUpdate &undo_record);
    virtual void UnUpdate(TermUpdate &undo_record);
};
```

Figure 1
Interface for class `Term`

```
class ExtendedTerm
{
public:
  // Constructor (from a plain Term)
    ExtendedTerm(Term* t);

  // Access
    char* Functor();
    void Selectors(StringSet &set)
    ExtendedTerm* Edge(char* selector);
    Term* Identity();  // return the "identity" of this equivalence class
    Term* Realize();   // return copy of current value as a Term

  // Unification
    void Union(ExtendedTerm* t, UndoStack &merges);
    void Deunion(UndoStack &merges);
    ExtendedTerm* Find();
};
```

Figure 2
Interface for class `ExtendedTerm`

```
class Context
{
public:
  // Constructor
    Context(Program* prog, Term* query);

  // Running the interpreter
    Boolean Eval();                // find next answer
    void PrintResult(File* out); // print query with bindings applied
    Term* Result();                // copy of query with bindings applied

  // Support for External Predicates
    void Mark();                   // "mark" the unify-undo stack
    Boolean unify(ExtendedTerm *, Term *);
    void ResetToMark();            // undo unifications done since "mark"
    Program* CurProgram();
};
```

Figure 3
Interface for class `Context`

```
typedef Boolean Predicate(Term* t);
void find_all(
      Predicate *filter,                          // acceptance criterion
      Program* prog,                    // pointer to program instance
      Term* query,                      // pointer to query
      TermList &answerlist)             // list to hold answers
{
      Context context(prog,query);
      while (context.Eval()) {
            Term* answer = context.Realize();
            if (filter(answer))
                  answerlist.Append(answer);
            else
                  delete answer;
      }
}
```

Figure 4
Calling the Congress Interpreter

```
dbclass UnixFile : public Term {
public:
  // override Term methods used by unification
    virtual void Selectors(StringSet &lab_set) { lab_set = EMPTY; }
    virtual Term* Edge(char* selector) { return nil; }
    virtual Boolean AddEdge(Term* )    { return false; }
    virtual Boolean RemoveEdge(char* ) { return false; }
    virtual char* Functor();           { return filename; }

  // Unix-File methods
    virtual int Size();
    virtual int Read(int offset, int length, char*  buffer);
    virtual Boolean Write(int offset, int length, char*  buffer);
private:
    dbchar* filename;
    dbvoid* contents;
};
```

Figure 5
A Term Representing a Unix File

```
dbclass Directory : public Term {
public:
    Directory(Directory* parent) : Term("dir")
        { contents.add(".",this); contents.add("..",parent); }
  // Term interface
    virtual Term* Edge(char* name)
        { return contents.lookup(name); }
    virtual Boolean AddEdge(char* name, Term* t) {
       if (contents.lookup(name) return false;
       contents.insert(name,t);
       return true;
    }
    virtual void Selectors(StringSet &lab_set)
        { contents.all_keys(lab_set); }
  // Unix directory methods
    Boolean link(char* name, Term* entry)
            { return AddEdge(name, entry); }
    Boolean unlink(char *name);
            { return RemoveEdge(name); }
      // etc
private:
    HashTable contents;
};
```

Figure 6
A Term that Emulates a Unix Directory

```
dbclass Procedure {
public:
    virtual char* Name();
    virtual Boolean Call(ExtendedTerm* query, ExtendedTerm* &continuation,
                         void* &workspace, Context* dynamic_state);
    virtual void Clean(void* workspace);
    virtual Boolean Backtracks();
};
```

Figure 7
Interface of class `Procedure`

```
Index DocsByTitle;

Boolean doc_title_call(ExtendedTerm* arg, void* &work, Context &context)
{
    Iterator* docs = (Iterator*) work;
    if (docs == nil) { // first time
        ExtendedTerm* title = arg->Edge("title");
        if (title != nil && !IsVariable(title))
            // A title was supplied, so use index
            docs = DocsByTitle->Lookup(title);
        else
            docs = DocsByTitle->Lookup(ALL_DOCS);
        work = (void*) docs;          // remember iterator for backtracking
    }
    Term* candidate = docs->Next();
    while (candidate) {
        context.Mark();               // mark unify undo stack
        if (context.unify(arg, candidate))
            return true;
        else {
            context.ResetToMark();    // undo unifications
            candidate = docs->Next();
        }
    }
    // no more candidates
    delete docs;
    return false;
}

// clean up code for doc_with_title EP
void doc_title_clean(void *work)
{
    delete (Iterator*) work;
}
```

Figure 8
`doc_with_title` External Predicate

4. Applications

To illustrate the power and flexibility of the POL approach, we describe three typical applications. The first example is a database of type and class information for C++ programs coupled with a language for specifying properties of classes and relationships between classes. The second application is a text document database with support for bibliographic citations. These two applications have been designed but not actually implemented. The third example is an actual working system, the attributed filesystem used in the CAPITL software development environment. It is outlined here and described in considerably more detail elsewhere [2].

4.1. C++ Design Constraints

The first example is inspired by a system called Clear++ [8]. In the authors' words,

C++ is an expressive language, but it does not allow software developers to say all the things about their systems that they need to be able to say. In particular, C++ offers no way to express many important constraints on a system's design, implementation, and stylistic conventions.

Clear is described as a greatly expanded version of the Unix tool Lint. Lint processes each source file of a C program looking for questionable constructs such as unreachable statements, type violations, and unportable code. It produces diagnostic messages and a file summarizing the type signatures of symbols imported and exported. A second pass compares these summary files looking for type mismatches between modules. As the authors of Clear++ point out, the flexibility of Lint is limited by the fact that the constraints it checks are wired in to the code. Thus, for example, it is not possible to check local coding conventions.

Clear++ translates C++ programs into an object-oriented database of summary information. Programmers annotate programs with constraints written in an *ad hoc* constraint language called CCEL, which is based on first-order predicate calculus. A separate tool applies these constraints to the database, looking for violations. Two examples constraints are

- *Every class name must begin with a capital letter.*
- *Every base class must have a virtual destructor.*[7]

The functionality of Clear++ would be easy to implement in POL. We would use the same database of summary information as Clear++. Each program component (class, member, function, etc.) is represented by an object. For example, a class definition is represented by an object of type `Class` with attributes including its name, list of superclasses, and list of methods. Instead of an *ad hoc* constraint language, we would simply use Congress. Database objects can be made accessible as Congress terms as described above. An example class definition, expressed as a term, might be

```
class(
    file=> "button.h"
    begin_line=> 140
    end_line=> 159
    name=> "CheckBox"
    super_classes=> [ TextButton, ... ]
    methods=> [ Press, ... ]
).
```

(The tags `TextButton` and `Press` represent references to other objects, of type `Class` and `MemberFunction`, respectively.)

Constraints written in Congress are easy to write and understand. The two constraints mentioned above can be easily expressed in Congress.

- *Every class name must begin with a capital letter.*

```
class_name_constraint(class(name=>N)) :-
    substr(string=>N, pos=>1, len=>1, substr=>I), is_capital(I).
```

[7]A base class is one with at least one subclass. The name of the destructor method of a class is the class name preceded by a tilde. Any method can be declared to be `virtual`. Subtle errors can occur if the `virtual` keyword is omitted in this context.

where `substr` and `is_capital` are external predicates for manipulating strings.

- *Every base class must have a virtual destructor.*

```
virtual_destructor_constraint(C:class) :-
    is_base_class(C), !, has_virtual_destructor(C).
virtual_destructor_constraint(C).

is_base_class(B:class) :-
    class(super_classes=>S), member_of(B,S).

has_virtual_destructor(class(methods=>M, name=>ClassName)) :-
    strcat(strings=>["~", ClassName], result=>DestrName),
    member(method(name=>DestrName, is_virtual=>true), M).
```

A query that prints all violations of the latter constraint is

```
?- class(C), not( virtual_destructor_constraint(C) ), print(C), fail.
```

An advantage to using a general-purpose system like POL, is that the information is available for other purposes beyond constraint checking. For example, someone trying to understand a complex program may want a list of all methods of a class, including methods defined in the class and methods inherited from superclasses. A query that produces such a list is easy to express in Congress.

```
all_methods(class(methods=>M, superclasses=>S), Result) :-
    add_methods(M,S,Result).

add_methods(M, [First | Rest], Result) :-
    all_methods(First, Inherited),
    union(M, Inherited, M1),
    add_methods(M1, Rest, Result).

add_methods(M, [], M).
```

where `union(L1,L2,Result)` merges the lists `L1` and `L2` eliminating duplicates.

4.2. Document Database

Computerized document formatting systems often come with tools for looking up citations and generating bibliographies. Examples under Unix include Refer, Bib, and BibTeX. They allow citations in the body of a document to be specified as a list of keywords. The tool scans the document, looking up each such citation in a bibliographic database. If a citation does not match any document, or if it matches multiple documents, the tool reports an error. Citations are replaced by annotations such as superscripts or bracketed numbers, and the cited references are collected and listed in a bibliography at the end of the paper. The bibliographic database is intended to be shared by many papers, perhaps written by different authors.

Musliner *et. al.* [20] have pointed out problems with these systems. First, the bibliographic "database" is not a true database. It lacks such database facilities as concurrency control, recovery, and distributed access. Second, references can become "stale" over time. A common problem experienced by users of these systems arises when they attempt to process a document they wrote some time ago. Because new items have been added to the database, citations that were perfectly fine when the document was written have become ambiguous. This problem can be avoided by saving the document with the citations already processed, but then the document will not track improvements in the database, such as error corrections, or updated references (for example, progression from "submitted for publication" through "to appear" to "September 1993, pp. 213-245").

A document formatting system built using POL could solve these problems. The persistent component of POL provides concurrent network access, locking, and recovery; hence sharing a database is easy. Congress, with its matching capabilities, is good for searching the database given imprecise information, and C++ is useful for state-based programming such as formatting, I/O, and index maintenance.

In such a system each document would be represented by a term with attributes `authors`, `title`, etc., as well as `contents` (the actual text of the document). Documents not stored in the database but referenced from it would missing the `contents` attribute. Citations could be represented as direct pointers to document objects, or as predicates, describing the intended target. For example, a database entry that would be represented in Refer as

```
%A Paul Adams
%A Marvin Solomon
%T An Overview of the CAPITL Software Development Environment
%J Fourth International Workshop on Software Configuration Management
%C Baltimore, MD
%D May, 1993
```

might appear in POL as

```
doc(
    title=>"An Overview of the CAPITL Software Development Environment"
    authors=> [
        person(last=>Adams, first=>Paul),
        person(last=>Solomon, first=>Marvin) ]
    contained_in=>SCM4,
    contents=>text(...),
    cites=>[
        citation(keywords=>["Ait","Kaci","LOGIN"]),
        citation(keywords=>["Adams", "Solomon", "CAPITL"]),
        ...
        ]
    ).
SCM4: conference(
    name=>"Fourth International Workshop on Software Configuration Management",
    short_name=>"SCM 4",
    publisher=>IEEE,
    date=>date(month=>May, year=>1993),
    location=>"Baltimore, MD."
    ).
IEEE: publisher( ... )
```

In this example the conference containing the paper is denoted by a pointer (indicated by the tag `SCM4`) to a separate term. Although not necessary, storing items such as conferences as separate, shared entities ensures that all documents will have consistent information, such as abbreviations and dates. Formatting tools could take advantage of this detailed information to support a choice of formatting styles. This mechanism is analogous to Bib string macros, but more structured and more flexible.

We have shown the citation information represented as a separate attribute. The contents could be created by an ordinary text editor with citations indicated as they are for Refer or Bib. (See the next section for a mechanism that allows existing text editors to access attributes of POL objects directly.) A simple tool would scan the contents of the document looking for citations and replacing them by indices into the `citations` list. Another tool could generate a bibliography from the citations. To avoid the problem of stale references, one could easily write a

Congress program that would check that each citation identifies a unique document, and add a pointer to that document to the `citation` term. Other forms of citations are also possible, including more complicated predicates than simple keyword matching.

As in the previous example, storing information in a POL database allows it to be used for other applications, such as a citation index. For example, a Congress query to find all the documents co-authored by "Paul Adams" and at least two other authors is easily specified.

```
?- D:doc(authors=>AList), member_of(person(last=>Adams), AList),
   AList = [A1,A2,A3|Tail], print(D), fail.
```

4.3. The CAPITL Object-base

We have used POL to build a database of objects or object-base as part of the CAPITL (Computer Aided Programming In The Large) [2] project. The CAPITL object-base can be considered an enhanced version of the Unix file system: More types of objects are supported, the set of attributes of an object is extensible, complex relationships among objects can be represented directly, and versioning of the entire database is efficiently supported.

Each object in CAPITL can be viewed as a "heavier weight" term that has certain fixed selectors with built-in semantics. Additional methods support Unix filesystem operations such as permissions, reading and writing. Other selectors are added by CAPITL to describe properties of each object necessary for software development such as an object's type.

CAPITL extends the set of POL base classes with special purpose internal node classes used to represent different filesystem objects. All CAPITL objects inherit from the class `Object` which contains the integer attributes `owner`, `group`, `permissions`, `mtime`, `atime`, and `ctime`, interpreted as in Unix, as well as `name` (the final component of the pathname) `directory` (the containing directory),[8] and `contents`.

Objects are further classified as *directories*, *symbolic links*, and *files*. A directory object is similar to a Unix file-system directory. Its `contents` attribute is a list of references to other objects. Its implementation is similar to the example in Section 3.2. The `contents` of a symbolic link is a pathname.

File objects are further classified as *plain*, *delta*, *term*, and *composite*. The `contents` of a plain file object is a byte-string atom. It has exactly the same semantics as a Unix "plain" file (see §3.2). Delta files have additional operations to "compress" and "uncompress" their contents. Delta files represent consecutive versions of their contents as delta lists using an algorithm similar to RCS [26]. The `contents` of a term file is an arbitrary Congress term. A composite file, like a directory, contains a list of references to other objects, but it does not emulate all the behavior of a Unix directory, nor is it constrained to be part of a strict tree structure.

CAPITL uses the versioning mechanism of POL to maintain multiple snapshots of the database. Each operation accessing the CAPITL database is done in the context of a designated *current world*, and any changes made by an operation affect only this world. A world is either *modifiable* or *committed* (read-only). There are mechanisms to choose a current world, commit a world, and spawn a new world as a child of an existing committed world.

There are two means to access the CAPITL object-base: an X-based browser and a Unix compatibility interface called *Emulated File System* (EFS) [24]. The X-based browser uses the C++ interface of terms to display CAPITL objects and to navigate the object-base. The browser supports visiting any object or directory and uses type-sensitive displays to depict the `contents` attribute of an object; other attributes are displayed using the Congress expression language (§2.4).

[8]CAPITL does not support the equivalent of Unix "hard" links. Thus each file has a unique name and a unique containing directory.

EFS allows programs to access CAPITL objects as if they were Unix files. It is based on the Network File System (NFS) facility [23], which is included in most versions of Unix. The EFS daemon *efsd* emulates an NFS server, treating a CAPITL database as a mounted file system, and allowing its objects to be manipulated by standard system calls (`open`, `read`, `write`, `seek`, `link`, `stat`, etc.) as if they were actual Unix files, directories, and symbolic links. Neither client programs nor the Unix kernel need be modified in any way. Thus standard editors such as *vi* or *emacs* can be used to edit the contents of file objects, and standard tools such as *cc* or *ld* can manipulate them.

Congress is used in CAPITL in two ways. As a query language, Congress allows objects to be selected by arbitrary predicates on their attributes, rather than forcing a strict hierarchical classification. As a programming language, Congress is used to construct tools that build and maintain configurations of software objects. The descriptive properties associated with CAPITL objects are used to ensure consistency of the configurations.

5. Related Work

There are three dimensions of Congress that interact: persistence, object-oriented programming and logic programming. While many researchers have concentrated on combining object-oriented and logic programming, or logic programming and persistence, few have tried to tie all three together. For completeness we consider some non object-oriented imperative languages when it is clear that their integration technique would work in an object-oriented setting.

One approach to combining logic and object-oriented programming is to add object-oriented features to a logic programming language. For example, Prolog might be extended with built-in operators for declaring classes and inheritance and explicit send and receive predicates for method invocation[10, 28]. An advantage of this approach is that it uses Prolog implementations, but it is unable to blend the procedural mechanisms of traditional object-oriented languages with the backtracking search of Prolog.

An alternative is to add logic programming features to an object-oriented language by extending the behavior of the object-oriented language[9] with Horn clause predicates [4, 12, 14]. This extension allows a tighter integration of the languages in which class methods may be written in a logic style.

A third alternative is to take two existing languages and create a bridge between them. The power of this approach is constrained by the bridge. For instance, the bridge described by Koschmann and Evans [16] implements function calls between LOOPS and Prolog, so LOOPS programmers can write methods in Prolog and Prolog programmers can access LOOPS objects. However, their bridge does not permit backtracking between the two languages and relies on the special hardware of a Lisp/Prolog machine.

Quintus Prolog [21] provides a software bridge to other programming languages, including C, Fortran, and Pascal. In Quintus Prolog programmers can call "foreign language procedures" (or FLPs) and pass terms as both in and out parameters. Library functions are used in the FLP to access terms and build new terms. For example in C, there are QP_put_xxx and QP_get_xxx for atoms, lists, and other Prolog data types. These functions are analogous to the methods of the class `Term` in Congress. The Quintus foreign language interface also allows FLPs to call the interpreter and receive solutions one at a time. Although quite powerful, their foreign language interface does not have the abstraction and encapsulation that an object-oriented language provides and hence programmers cannot extend the behavior of terms using inheritance.

[9]C++ does not allow such extensions without changing the compiler; languages such as Smalltalk-80 or LOOPS allow a programmer to add new meta-classes, which is analogous to altering the interpreter for a language.

Our embedding lies between the bridge approach and a complete integration of logic programming with C++. The embedding of Congress in C++ is similar to the FLPs of Quintus: Instances of Congress classes such as `Program`, `Procedure`, and `Context` form a bridge to C++ via their methods; C++ programmers can call the interpreter to solve queries. In addition, however, the inheritance of C++ allows a term interface to be added to existing data structures enabling access from Congress programs.

Through its use of E, POL gets the database features that Exodus provides: persistence, identity, and concurrency control. None of the other approaches for combining object-oriented and logic programming have these features.

Persistence has been added to logic programming languages in many different forms. One approach focuses on the logic capabilities of the language and usually restricts the data types to avoid function symbols. Here the key idea is to extend relational databases with inferencing. A good survey of this approach can be found in Gallaire *et al*[11]. Simpler approaches use a relational database as a repository for facts and couple that repository with an existing Prolog implementation [13]. Our approach is closest to that of Moffat and Gray [19] who implement Prolog in the persistent language PS-ALGOL[3]. We differ from them by giving external predicates the possibility of backtracking.

6. Status and Future Work

A prototype of POL is working and has been tested extensively during the construction of CAPITL. POL's extendible data model, versioned term space, and embedded logic programming language provided a good platform on which to explore the logic-based approach to software development used by CAPITL.

The ideas in POL originally grew out of the database and programming needs of CAPITL, a purpose which it has served well. We are intrigued by the potential of applying the POL approach to a much wider range of applications. Two examples are outlined in Sections 4.1 and 4.2. These applications have been designed but not yet implemented. We need to produce working prototypes of these applications to verify the practicality of the approach and search for additional applications to explore its generality.

In addition to finding new applications, there are number of extensions that would make POL better and easier to use.

Performance. The speed of Congress programs appears to be a bottleneck in CAPITL. Known compilation techniques and interpreter optimizations for Prolog have not yet been applied to Congress. Adding some of these techniques to Congress is "simply a matter of programming." Others may conflict with the circular unification and object identity that are essential features of Congress. Additional research may be needed before they can be applied.

Congress Programming Environment. Currently a Congress programmer has little support for constructing working programs. There is a primitive trace facility, but it is verbose and awkward. A simple profiler that counts procedure calls is also available. We would like to add an interactive debugger and better profiling capabilities. Better support for writing EP's is also needed.

Updates Although updates from Congress programs have been used successfully in CAPITL, they require significant programming skill to be used effectively. One problem is the current update operator does not have a "logical" semantics. We are exploring other approaches, as indicated in Section 3.4.

Logical Semantics The semantics of Congress are specified operationally. We would like to construct a denotational semantics, perhaps based on minimal-model techniques. Some of the groundwork has been laid by Hassan Aït-Kaci. Techniques from modal logic as suggested by Manchanda and Warren may be appropriate to modelling assignment.

C++ Methods Congress programmers can call C++ methods by writing an external predicate for each method call. Such EP's are simple to write, but tedious. We would like to extend the syntax of Congress to support method calls and add to the compiler the ability to generate the necessary EP's automatically.

7. References

[1] H. Aït-Kaci and R. Nasr, LOGIN: A Logic Programming Language with Built-In Inheritance, *Journal of Logic Programming*, Mar. 1986, 181-215.

[2] P. Adams and M. Solomon, An Overview of the CAPITL Software Development Environment, *Fourth International Workshop on Software Configuration Management*, Baltimore, MD, 1993. Also available as University of Wisconsin—Madison Computer Sciences Technical Report 1143, April 1993.

[3] M. P. Atkinson and O. P. Buneman, Type and Persistence in Database Programming Languages, *Computing Surveys 19*, 2 (June 1987), 105-190.

[4] T. A. Budd, Blending Imperative and Relational Programming, *IEEE Software 8*, 1 (Jan. 1991), 58-65.

[5] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita and S. Vandenberg, The EXODUS Extensible DBMS Project: An Overview, in *Readings in Object-Oriented Databases*, S. Zdonik and D. Maier (ed.), Morgan-Kaufman, 1990.

[6] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, NY, 1984.

[7] J. R. Driscol, N. Sarnak, D. D. Sleator and R. E. Tarjan, Making Data Structures Persistent, *Journal of Computer and System Sciences 38*, 1 (Feb. 1989), 86-124.

[8] C. K. Duby, S. Meyers and S. P. Reiss, CCEL: A Metalanguage for C++, *USENIX C++ Workshop*, 1992.

[9] M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley, Reading, Massachusetts, 1990.

[10] K. Fukunaga and S. Hirose, An Experience with a Prolog-based Object-Oriented Language, *OOPSLA '86 Proceedings*, Sep. 1986, 224-231.

[11] H. Gallaire, J. Minker and J. Nicolas, Logic and Databases: A Deductive Approach, *Computing Surveys 16*, 2 (June 1984), 153-185.

[12] M. H. Ibrahim and F. A. Cummins, KSL/Logic: Integration of Logic with Objects, *IEEE International Conference on Computer Languages*, Mar. 1990, 228-235.

[13] Y. E. Ioannidis and M. M. Tsangaris, The Design, Implementation, and Performance Evaluation of Bermuda, University of Wisconsin-Madison Tech Report #973, Oct. 1990.

[14] Y. Ishikawa and M. Tokoro, A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation, *OOPSLA '86 Proceedings*, Sep. 1986, 232-241.

[15] K. Knight, Unification: A Multidisciplinary Survey, *ACM Computing Surveys 21*, 1 (Mar. 1989), 93-124.

[16] T. Koschmann and M. W. Evens, Bridging the Gap between Object-Oriented and Logic Programming, *IEEE Software 5*, 4 (July 1988), 36-42.

[17] S. B. Lippman, *C++ Primer, Second Edition*, Addison Wesley, Reading, Massachusetts, 1991.

[18] S. Manchanda and D. S. Warren, A Logic-based Language for Database Updates, in *Foundations of Deductive Databases*, J. Minker (ed.), Morgan-Kaufmann, Los Altos, CA, 1987, 363-394.

[19] D. S. Moffat and P. M. D. Gray, Interfacing Prolog to a Persistent Data Store, in *Proceedings of the Third International Conference on Logic Programming, Lecture Notes in Computer Science*, vol. 225, Springer-Verlag, July 1986, 577-584.

[20] D. J. Musliner, J. W. Dolter and K. G. Shin, BibDb: A Bibliographic Database for Collaboration, *CWSC '92 Proceedings*, Oct. 1992, 386-393.

[21] *Quintus Prolog, Release 3.1*, Quintus Corporation, An Intergraph Company, Palo Alto, California, Revised July 1991.

[22] J. Richardson, M. Carey and D. Schuh, The Design of the E Programming Language, *ACM Trans. Prog. Lang. and Systems* , to appear.

[23] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, Design and implementation of the Sun Network filesystem, *Proceedings of the Summer 1985 USENIX Conference*, Portland, OR, June 1985, 119-130.

[24] M. Solomon, EFS: The Extensible File System, University of Wisconsin Technical Report, In preparation.

[25] B. Stroustrup, *The C++ Programming Language, Second Edition*, Addison Wesley, Reading, Massachusetts, 1991.

[26] W. F. Tichy, RCS: A System for Version Control, *Software—Practice and Experience 15*, 7 (July 1985), 637-654.

[27] D. S. Warren, Database Updates in Pure Prolog, *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984, 244-253.

[28] C. Zaniolo, Object-Oriented Programming in Prolog, *1984 International Symposium on Logic Programming*, Feb. 1984, 265-270.